A FRAMEWORK FOR AN AUTOMATED COMPILATION SYSTEM FOR RECONFIGURABLE ARCHITECTURES

THESIS

George R. Roelke IV
2nd Lieutenant, USAF

AFIT/GE/ENG/97M-01

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GE/ENG/97M-01

A FRAMEWORK FOR AN AUTOMATED COMPILATION SYSTEM FOR RECONFIGURABLE ARCHITECTURES

THESIS

George R. Roelke IV
2nd Lieutenant, USAF

AFIT/GE/ENG/97M-01

AFIT/GE/ENG/97M-01

A FRAMEWORK FOR AN AUTOMATED COMPILATION SYSTEM FOR RECONFIGURABLE ARCHITECTURES

THESIS

Presented to the Faculty of the Graduate School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Computer Engineering

George R. Roelke IV

2nd Lieutenant, USAF

March 1997

Preface

The purpose of this thesis was to investigate the problem of automating the development of applications for reconfigurable computers. This was accomplished by creating a framework for a high-level language based development system, determining the key tasks involved, and creating a test implementation of part of the system.

I would like to thank all of the people who have given me advice, support, and encouragement throughout the thesis process. I would like to thank my thesis advisor, Dr. Henry Potoczny, for providing guidance and encouragement, and for helping me see the obvious. I also owe a great deal of thanks to my (other) advisor, Lt.Col. Dave Gallagher. Although I was not technically his thesis student, he spent countless hours with me discussing the problems, processes, and solutions contained in this document. And along the way, he even managed to give a few Air Force insights to this brand new Second Lieutenant.

I would also like to thank the other member of my committee, Lt.Col. Tom Wailes, for providing several much needed comments, and pointing out one or two areas of additional research. In addition, I must thank the sponsor of this research, Keith Anthony of the National Air Intelligence Center, for providing a fascinating topic, and for allowing me the flexibility to choose my own path into this wide-open area of research.

Finally, I owe thanks to everyone who has helped me stay sane throughout my 18 months at AFIT. I'd like to thank James Savage, Javier Marti, Bruce Hunt, and everyone

else in the VLSI lab, whose humor lightened up more than one dark and stressful day.

And most of all, I'd like to thank my parents and my brothers, for providing the continued love, understanding, and support that have helped carry me to where I am today.

George Roelke

Table of Contents

List of Figures

List of Tables

Abstract

The advent of the Field Programmable Gate Array has allowed the implementation of runtime reconfigurable computer systems. These systems are capable of configuring their hardware to provide custom hardware support for software applications. Since these architectures can be reconfigured during operation, they are able to provide hardware support for a variety of applications, without removal from the system. The Air Force is currently investigating reconfigurable architectures for avionics and signal processing applications.

This thesis investigates the problem of automating the application development process for reconfigurable architectures. The lack of automated development support is a major limiting factor in the use of these systems. This thesis creates a framework for a reconfigurable compiler, which automatically implements a single high level language specification as a reconfigurable hardware/software application. The major tasks in the process are examined, and possible methods for implementation are investigated. A prototype reconfigurable compiler has been developed to demonstrate the feasibility of important concepts, and to uncover additional areas of difficulty.

A FRAMEWORK FOR AN AUTOMATED COMPILATION SYSTEM FOR RECONFIGURABLE ARCHITECTURES

I. Introduction

1.1 Background

Historically, several approaches have been available to designers of electronic applications. The primary differences between these approaches are specialization and flexibility. By specializing the design, very good performance for a particular application can be achieved. Unfortunately, a more specialized circuit is much more difficult to adapt to perform other applications. Designers often must make a decision between performance and flexibility.

Approaches to circuit design range across a wide spectrum. At one extreme, Application Specific Integrated Circuits (ASICs) are designed to implement a single application with the best performance possible. These circuits usually cannot be used for other tasks. At the other extreme, general purpose processors (GPPs) are designed to implement a range of applications by providing a flexible instruction set. Complicated functions can be constructed from these instructions. Unfortunately, the instruction set is by necessity very broad, and many functions which have very simple hardware implementations cannot be included. For example, the bit-reversal operation, in which the order of the bits in a register is reversed, is very useful in many digital signal processing (DSP) applications. This operation is not implemented in most general purpose processors, and must be implemented with some sequence of other instructions.

The result is a much slower implementation of the bit-reversal operation than could be achieved by custom hardware. As a result, the performance of a GPP is typically much less than that of an ASIC with custom hardware support for any particular application.

A third method, the Application Specific Instruction Processor (ASIP), is an attempt to provide a compromise between the performance of an ASIC and the flexibility of a general purpose architecture. ASIPs add specialized instructions to a GPP to improve its performance for a specific type of application. For example, Digital Signal Processors (DSPs) provide most of the flexibility of a GPP, while providing additional hardware support for common DSP operations, such as the multiply-accumulate and bit-reversal instructions.

In the last several years, a fourth alternative has drawn increasing attention from researchers. Reconfigurable computer architectures provide a means of achieving performance close to that of ASICs on certain applications, while maintaining the flexibility of more general purpose hardware. Sometimes called Custom Computing Machines, reconfigurable architectures can be configured to provide custom hardware support for one application, and then reconfigured to support additional applications. The Air Force is currently investigating this technology as a means of providing high performance signal processing hardware to combat aircraft. Current approaches utilize expensive ASICs which must be replaced whenever the DSP algorithms are upgraded. Reconfigurable architectures can be reprogrammed to implement these new algorithms with only a software upgrade, without removal from the aircraft or the purchase of new

hardware. This results in a tremendous savings in both hardware cost and in aircraft down-time, since the upgrade does not require the aircraft to be removed from service.

Reconfigurable computer architectures are based upon programmable logic devices (PLDs). These circuits can be programmed by the user to implement both combinational and sequential logic. PLDs can be used to implement the same digital logic as an ASIC, and they can be modified after fabrication. Several families of programmable devices exist. Some of these devices are programmable only once (such as fuse-based PROMs), while others can be modified many times. Until the development of SRAM-based Field Programmable Gate Array (FPGA) technology in the mid 1980s, all of these devices required the removal of the chip from the system during programming. FPGAs can be reprogrammed in-system, without the use of ultra-violet light or higher-than-normal voltages. These devices form the basis for a growing class of systems which can modify their hardware to suit changing applications.

Reconfigurable devices allow for hardware implementation of functions which would otherwise be done in software using a general purpose processor. If the application is subsequently changed, the devices can be reconfigured to meet the needs of the new application. One model for such a system is a reconfigurable coprocessor attached to a general purpose processor. Examples of these systems include Wright Laboratory's CHAMP (Configurable Hardware Algorithm Mappable Processor) system [11] and Brown University's PRISM system [14]. Research with similar systems has

shown speedups ranging from 20 to 1000 times over GPP implementations for a variety of applications, from genetic algorithms to digital signal processing [5:17].

Two major limiting factors to reconfigurable systems are device density and development tools. Because of the overhead required to implement logic reconfiguration, FPGAs provide only a fraction of the number of gates an ASIC can provide on a single chip. Currently, the largest FPGAs have a gate capacity in the 50-100K gate range. FPGAs have been increasing in density rapidly, however, and it is predicted that gate densities will approach one million by the turn of the century.

A more significant problem is the lack of software tools to support the development and use of reconfigurable systems. With current tools, hardware description languages (such as VHDL or Verilog) or gate-level schematic entry tools must be used to develop the logic for reconfigurable applications. In the coprocessor model, the code for the GPP must be developed separately and manually interfaced with the reconfigurable system. Applications developers must have detailed knowledge of the hardware and the ability to design digital hardware. For reconfigurable systems to be adopted on a larger scale, automated tools must be developed which enable those without engineering knowledge to use them.

1.2 Problem

The current method of application development for reconfigurable systems typically requires the use of a hardware description language. While these languages

describe digital logic quite well, they lack the general purpose functionality of a high level programming language such as C or Ada. In addition, the number of engineers who can use VHDL or Verilog is very small compared to the number of software developers who use high level languages. There are many applications written in high level languages for general purpose computers which could benefit from implementation on reconfigurable systems. Unfortunately, the tasks of identifying which parts of a HLL application can be done better with hardware support and creating the mixed hardware/software specification is not automated, and requires a great deal of time and effort from skilled engineers.

High level language compilers remove the requirement for detailed knowledge of the underlying computer hardware and assembly language in conventional systems. Researchers are now beginning to develop similar compilers for reconfigurable machines. A compiler for a reconfigurable computer would take an application written in a single specification and identify those parts of the program which would benefit most from hardware support. The compiler would create the hardware and software automatically, removing the need for the developer to understand digital logic design or the underlying hardware. Such compilers would greatly reduce development time and costs for reconfigurable applications.

Unfortunately, the problem of compiling high level language code to execute on a reconfigurable system is more complicated than for a conventional system. Conventional processors have fixed instruction sets. Compilers for these systems must fabricate the

more complex operations required in an application from these building blocks. Reconfigurable systems, however, can be modified to implement portions of the application in hardware, leaving the rest as conventional code for a GPP. The goal is to move those portions of code which can be done faster in hardware away from the GPP. Unfortunately, identifying those portions can be very difficult. Since HLLs provide constructs and operations based upon general purpose architectures, they do not provide concise ways of describing operations which do not exist on typical GPPs. Many operations which have direct hardware implementations must be described by long sequences of simple instructions. Recognizing that a sequence of instructions can be replaced by a single hardware structure is difficult.

Once those operations which can be implemented in hardware have been identified, the compiler must select the most suitable candidates. This process depends heavily on the characteristics of the target hardware. Since FPGAs are limited in the amount of logic that can be implemented, only a fraction of these candidates can be mapped to hardware at any given time. In addition to the raw computation time, overhead due to communication, memory accesses, and configuration time can sometimes result in a hardware execution time that is longer than the original software version. Only those blocks which will execute faster in hardware should be moved. Proper estimation of these factors is critical to the correct partitioning of an application.

The problem of concurrently designing combined hardware/software applications is often referred to as Hardware/Software Codesign. The hardware and software

specifications are developed simultaneously, carefully considering the tradeoffs associated with implementing different portions of the application in hardware or software. The primary focus of codesign research has been for embedded control applications, augmenting microcontrollers with custom hardware to meet real-time operating constraints that could not be satisfied with a software-only system. The hardware portion is typically implemented with ASICs, although FPGAs are coming into increasing use. However, the FPGAs are programmed only once, and are not modified during operation. Most of these methods use HDLs or other specialized languages (e.g. object-oriented specification languages, UNITY, etc.) to allow the developer to manually describe the partitioning between hardware and software. A great deal of effort is still needed to create a combined hardware/software application.

Some research has been done to develop techniques to automate the partitioning of a single specification between hardware and software. However, the focus of this research has been on static logic (whether it is implemented with ASICs or FPGAs) and does not consider benefits and limitations unique to dynamically reconfigurable hardware. For example, with static systems, configuration time is not a factor in the partitioning estimation process. In addition, hardware cost estimation changes significantly when dynamically reconfigurable systems are used. Reconfigurable systems usually have tighter limits on the amount of logic that can be implemented at any instant, but system can be reprogrammed any number of times during the application (effectively providing infinite hardware). It is also more difficult to predict whether a design with a certain gate count can be placed onto a FPGA, due to the difficulties in routing on

FPGAs. This difference in focus between conventional hardware/software codesign and reconfigurable codesign can be quite significant.

This thesis proposes a framework for an automated application development system for reconfigurable architectures. This system, called a reconfigurable compiler, uses a single high level language specification to create a mixed software/hardware application that takes full advantage of the runtime reconfiguration capabilities of the target architecture. The major tasks of the reconfigurable compiler are identified and examined. The key factors used in deciding which portions of the code to implement with hardware are discussed, and methods for their estimation are proposed.

1.3 Assumptions

The primary goal of this thesis is to investigate the problem of creating a compiler for reconfigurable computer systems. Since there are many different architectures under investigation, the results should be independent of the hardware model as much as possible. A general loosely-coupled coprocessor model is used as the basis for later assumptions, however, since this architecture is the most similar to modern computers. Coprocessor systems provide the most obvious target for mixed hardware/software applications, and would benefit most from compiler support. Since estimates must be made for communication times, gate capacities, and other characteristics of reconfigurable systems, some attempt has been made to specify typical characteristics of current systems.

1.4 Objectives

There are two primary objectives for this research. The first objective is to identify the unique problems which must be overcome to create a compiler for reconfigurable computers. The major tasks facing a reconfigurable compiler are detailed, and methods for accomplishing each of these tasks are suggested. The second objective looks in greater detail at two of these tasks, the identification and selection of hardware mappable portions of conventional C code. Some blocks of code cannot be implemented in hardware for a variety of reasons, while others would simply be unprofitable, due to the amount of hardware required, the communication time, or other factor. Methods for estimating the important characteristics necessary for the proper partitioning of code between hardware and software are suggested.

1.5 Approach/Methodology

To provide a basis for this work, a literature search was conducted in the areas of reconfigurable computer architectures, compilers, hardware synthesis, FPGA technology forecasts, hardware/software codesign, and VHDL. An analysis of current reconfigurable systems was performed to select an architectural model to upon which to base later design decisions. Predictions of gate density and speed increases in FPGAs are used to predict hardware limitations.

Methods of hardware synthesis were examined to determine how hardware mappable portions of C code could be identified. While there has been a significant amount of work in the area of behavioral VHDL synthesis, some adaptation is necessary

to use these methods for C. The area of hardware/software codesign was examined to determine how applications are partitioned for embedded system design, and how that knowledge could be applied to reconfigurable systems. Most of the criteria used in the partitioning of codesign applications have counterparts in the reconfigurable area, but additional criteria must be introduced to reflect several unique differences. Methods for estimating all of these criteria are suggested.

To demonstrate the concepts introduced in this work and to gain problem insights which only become apparent when one actually attempts to implement a solution, an optimizing C compiler was modified to implement the estimation and extraction stages of the reconfigurable compiler. The implementation examines the source code for feasible code blocks, makes the required performance and cost estimates for each block, and outputs the estimates for use by a partitioning program. After the partitioner selects which block should be extracted for hardware implementation, the reconfigurable compiler removes the marked code blocks, inserts interface code, and outputs the software and hardware specifications as standard C code. The software portion is compiled normally, while the hardware code can be used to create a behavioral HDL description which would ultimately be synthesized. The software system was evaluated using specially created code and the code from an Infrared Missile Warning System developed for the CHAMP board at Wright Laboratory.

II. Background

2.1 Introduction

This chapter provides an overview of reconfigurable computer architectures and the tools available for application development. Programmable logic devices, the basis for reconfigurable architectures, will be examined, providing an overview of their current capabilities and limitations. The three major architectural models are discussed, and several example systems are introduced. The remainder of the chapter focuses on the application design process for reconfigurable systems. An overview of the design process is given, followed by examinations of the four major approaches to development: schematic entry, conventional hardware description languages, custom hardware description languages, and high level language extraction. The latter method, high level language extraction, is the most relevant to this thesis, and past and current work in the area is examined in some detail.

2.2 Programmable Logic Devices

2.2.1 Overview

The term programmable logic device is used to describe electronic devices which can be configured by the user to implement arbitrary digital logic. Programming can be accomplished either at the foundry or in the field (by the user). The latter family of devices is typically referred to as Field Programmable Logic Devices (FPLDs). These devices can be programmed by the user, with little or no special equipment. Depending on the type of device, an FPLD can be programmed only once or many times.

An important type of FPLD is the Field Programmable Gate Array (FPGA). The FPGA provides the basis for current research in reconfigurable computers. Xilinx marketed the first FPGA in 1985 [52:1]. FPGAs are based upon static random access memory (SRAM) look-up tables and programmable interconnect. A typical FPGA provides a large number of configurable logic blocks (CLBs), which are derived from SRAM look-up tables. A 16 entry look-up table containing one output bit can implement any combinational logic function containing up to four inputs. Different manufacturers provide FPGAs with different granularity.

The CLB structure is typically augmented with one or more flip-flops, and sometimes with other common digital structures which would otherwise have to be implemented using a number of CLBs (typically one or more small multiplexors and fast carry generation logic for use in adders, etc.)[52:2-9]. Increasingly, some amount of on-chip RAM is provided as well. Typical FPGAs include several hundred to several thousand CLBs arranged in a two-dimensional mesh. A fragmented network of wires fills the space between the CLBs, as shown in Figure 1. Connections between two CLBs can be made by connecting the output of one CLB to one of the wires of the interconnect, through the interconnect, and finally to the input of the second CLB. Most of the lines in the interconnect are very short, so routing between two distant CLBs often requires the connection of several lines. Connections are made by transmission gates, whose switch inputs are determined from SRAM configuration registers.

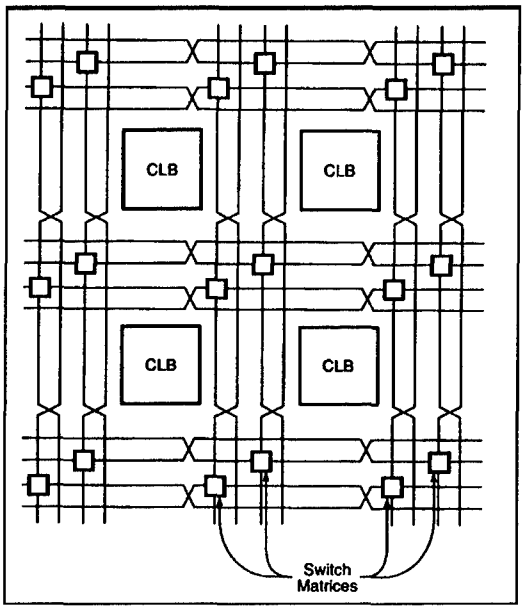

Figure 1: Typical FPGA Structure [52]

2.2.2 FPGA Configuration

The configuration information of an FPGA is called a bitfile. The bitfile contains the configuration information of each of the CLBs and interconnect devices on the FPGA, and is essentially an image of all of the SRAM configuration elements on the chip. Most FPGAs are configured serially, although newer devices allow a limited parallel input (either 8,16, or 32 bits) for faster configuration [20][36]. Configuration of an FPGA can be accomplished in several microseconds to several milliseconds, depending on the size of the FPGA and the fraction of the FPGA being reprogrammed. Some FPGAs can be partially reconfigured, allowing a portion of the FPGA to be configured while the remainder of the FPGA is still functional. Most FPGAs, however, require that the entire FPGA be reconfigured at once.

An excellent example is the Xilinx XC4025 FPGA. The XC4025 contains 1,024 CLBs, providing roughly 25,000 gates of usable logic [52:2.26]. Roughly 350 bits of configuration information are required for each CLB and its neighboring interconnect. With error detection bits and other additional overhead, a total of 422,168 bits are required for configuration data. The XC4025 can be configured either serially, or in parallel using an 8 bit bus. The configuration clock speed of the device is roughly 10MHz, so the configuration time of the entire chip is roughly 5.3ms.

2.2.3 FPGA Advantages

FPGAs possess several characteristics which make them useful for reconfigurable computing. The largest advantage is reconfigurability. Since FPGAs do not require special programming voltages or hardware, so they can be reconfigured in the system. This allows hardware reconfiguration under the control of the application itself. Complete configuration of even the largest FPGAs can be accomplished in milliseconds. Since FPGAs are based upon SRAM, they can be reconfigured an unlimited number of times. This results in tremendous flexibility. FPGAs can implement any digital logic circuit, limited only by the size of the device. When applied to reconfigurable computing, application developers can implement arbitrary hardware circuits at runtime and change those circuits as often as desired.

2.2.4 FPGA Limitations

There are several difficulties associated with the use of FPGAs. The SRAM structure which gives FPGAs their unique capabilities creates some special limitations.

The most important limitations of FPGAs are density and speed. A great deal of die area is needed by the interconnect lines, the configuration registers, and the CLBs. This overhead severely reduces the amount of logic an FPGA can implement. While a conventional ASIC can provide well over a million logical gates, the largest FPGAs provide only 100,000 gates. In addition, the long interconnect lines can create large delays, and result in circuits that are slower than ASIC implementations. The fastest FPGA circuits operate at 50MHz or less. These limitations may be critical to developers who must choose between an ASIC implementation and an FPGA implementation of static logic. In the area of reconfigurable computing, however, the performance offered by a FPGA may provide a substantial improvement over a software implementation of an algorithm.

2.2.5 Estimates of Future Capabilities

FPGA designs continue to evolve to better serve reconfigurable system developers. Many of the limitations of current FPGA designs will be reduced in future FPGAs. The most obvious limitations, density and performance, will be reduced as CMOS technology improves. FPGAs will always provide fewer gates and lower performance than ASICs, but as density increases, larger and larger applications will be achievable. Fawcett estimated in 1995 that FPGAs would provide over 100,000 gates (which has already been reached in 1996) and operate at 200MHz by the end of the century [20:160]. The capacity of FPGAs will continue to increase in direct relation to the increases in VLSI device density.

Just as important as density increases, however, are architectural changes. One of the current limitations of many FPGAs is configuration time. While the time to configure an entire FPGA is very small (<10ms) for many uses, it is an extremely long period of time to a computer system with a clock period measured in single nanoseconds. Some current FPGA architectures allow partial reconfiguration, while others will provide that capability in the near future. Many FPGAs are moving away from the serial configuration used in the past, allowing 8, 16, or 32 bit parallel input. This is ideal for configuration over a microprocessor bus.

Another suggested FPGA architecture is the Dynamic Programmable Gate Array (DPGA) [17]. The DPGA is based upon a standard FPGA, augmented with additional configuration registers to allow fast switching between two or more configurations. Once the configurations are loaded, switching between configurations requires only a single clock cycle. Internal state can even be saved. This will allow easier task switching and thus multi-tasking of hardware. This architecture would make FPGA hardware more useful to applications in a multi-tasking environment.

Recognizing that many FPGAs are being interfaced with standard computer systems, Xilinx is developing an architecture specially adapted for use in these systems [20:164-165]. Most FPGA designers must create the bus interface hardware manually, utilizing the limited number of CLBs. The XC6200 series devices have dedicated interface circuitry for this task, offering better performance with less die area. The CLB structures have been made less complex and more numerous, providing a more regular

structure that is easier to route. The XC6200 also supports partial reconfiguration, requiring only 3 bytes of information per logic cell. As a result, programming of the chip is much faster than in previous devices, allowing configuration in only 200µs. As proposed by the DPGA architecture, the XC6200 will support multiple configurations and rapid swapping. The state of internal logic will be saved and can be restored very quickly. It is likely that many of these characteristics will be incorporated into future reconfigurable computers.

2.3 Reconfigurable Computer Architectures

A reconfigurable computer is a computer system which can modify the characteristics of its hardware to provide increased performance for a specific application. These systems are typically based upon Field Programmable Gate Arrays (FPGA), which can be configured by the user to implement a variety of digital logic functions. In a conventional system, the hardware configuration remains constant, providing a fixed set of capabilities to the user. Reconfigurable architectures allow a developer to optimize the hardware for a particular application, while maintaining the flexibility to quickly adapt the system to other applications. In effect, reconfigurable architectures allow a developer to achieve some of the performance gains of special purpose hardware, but retain the benefits of a general purpose machine.

There has been a limited amount of research into this field. Until ten years ago, programmable logic devices were too limited in density and performance to allow the creation of useful reconfigurable systems. The older programmable devices, such as the

Programmable Logic Array (PLA) and the Programmable Read-Only Memory (PROM), could provide only a small amount of logic. In addition, these devices could not be modified after programming without removing them from the system. In 1985, Xilinx introduced the Field Programmable Gate Array. While initially limited in speed and density, FPGAs have shown a 35%-per-year improvement in speed, and a better than 55%-per-year improvement in density [52:1]. FPGAs capable of implementing the equivalent of 100,000 gates of logic are currently in production.

A range of reconfigurable systems have been built in recent years, by researchers and commercial developers. Most of these systems fall into three major categories. The primary difference between the categories is the method in which the reconfigurable hardware is used. Other differences include the method in which the reconfigurable hardware communicates with attached general purpose or custom processors, and in the level of reconfigurability. The following sections provide an overview of the three families of reconfigurable architectures, and give several examples of each type.

2.3.1 The Static Logic Model

The first systems developed using reconfigurable logic were based upon the static logic model. In this model, the FPGAs are programmed upon system or application startup and are not reconfigured during operation. The reconfigurable hardware can be viewed as any other fixed hardware device, whose specific functionality is determined before operation begins. As a result, the amount of logic a system can implement is limited to the size of the FPGAs. Typically, only a few thousand gates are implemented

on these systems. Since the FPGAs are not reprogrammed during operation, the overhead incurred by configuration is paid only once and does not slow operation. The reconfigurable hardware can be either standalone or a part of a larger system.

The reconfigurable hardware in a static system is typically less tightly integrated into the overall system than in the other families. When it is connected to a conventional computer system, the reconfigurable hardware is typically not closely linked to the system. Most static logic systems communicate to the host processor via the host's I/O interface or system bus. The reconfigurable hardware is used to perform a single function. Information is sent to the reconfigurable hardware, the function is performed, and the result is copied back. The reconfigurable system may or may not be allowed to access system memory for this purpose. Figure 2 shows a typical configuration for the static logic model.

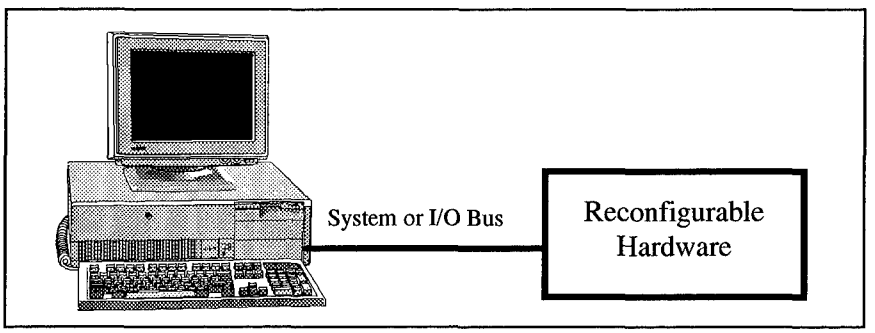

Figure 2: The Static Logic Model

An example of a static logic system is the PeRLe system, developed by DEC Paris [10]. The PeRLe system was based upon a 5x5 array of Xilinx 3090 FPGAs, providing roughly 150,000 gates to the user. Applications developed for the system include RSA cryptography, LaPlace transforms, long multiplication, and a stereoscopic vision system.

In 1990, PeRLe set a speed record for RSA cryptography using 512-bit keys, delivering roughly ten times the performance of a custom VLSI implementation [20:157]. This performance was achieved largely as a result of the ability to easily customize the algorithm on the reconfigurable hardware.

Other examples of static logic systems include the SPLASH processor and the ACE-12 system. SPLASH was developed at the Supercomputing Research Center [22]. SPLASH-1 was constructed of 32 FPGAs linked in a linear array, connected via a VME bus to a Sun workstation. SPLASH was designed to implement systolic algorithms for one-dimensional pattern matching in DNA research. It outperformed a Cray-2 by a factor of 325, and a custom built ASIC device by a factor of 45 [20:158]. An example of a commercial system is the ACE-12 system, developed by Metalithic Systems. The ACE-12 contains up to 12 FPGAs, using a SIMD architecture for processing. A swap/sort algorithm implemented on the ACE-12 system executes more than 360 times faster than on an Intel 486/33 processor.

The static logic model is the most similar to current ASIC-based systems, and thus most of the first reconfigurable systems were based upon it. While many static logic machines have been able to show great performance gains over GPP and sometimes ASIC implementations, they fail to take advantage of many of the key features of programmable logic. The primary advantage of these systems over ASIC implementations is a faster development time, and the ability to rapidly "tweak" the algorithm to incorporate changes made as a result of execution profiling.

2.3.2 The Coprocessor Logic Model

2.3.2.1 Description

A more flexible model of reconfigurable systems is the coprocessor model. In these systems, the reconfigurable logic supplements a conventional processor. This model is sometimes referred to as the *loosely-coupled model*, since the reconfigurable coprocessor is more closely integrated with the main processor than in static systems (which may not be attached to one at all), but not as closely as in the dynamic instruction set model (discussed in the next section). Portions of an application which can be implemented quickly and efficiently in hardware are partitioned to the FPGAs, while the remaining portions are executed on a general purpose processor. The processor can reconfigure the FPGAs either before execution of an application begins or during execution.

Many examples of this type of architecture exist, including the CHAMP system [11], the PRISM system [5][1], the Transmogrifier system [21], and Virtual Computer Corporation's EVC-1 [16]. These systems are comprised of one or more FPGAs connected to a host processor. They typically provide between 4,000 and 400,000 logic gates to the user. The system is configured either directly by the application or through operating system calls. For those systems which allow the application to reconfigure the FPGAs during execution, reconfiguration time becomes very important to the overall speed of the application. As discussed in Section 2.2.2, configuration time is typically on

the order of several milliseconds, a very long time for modern CPUs running with clock periods of less than 10 ns. Figure 3 shows a typical coprocessor architecture.

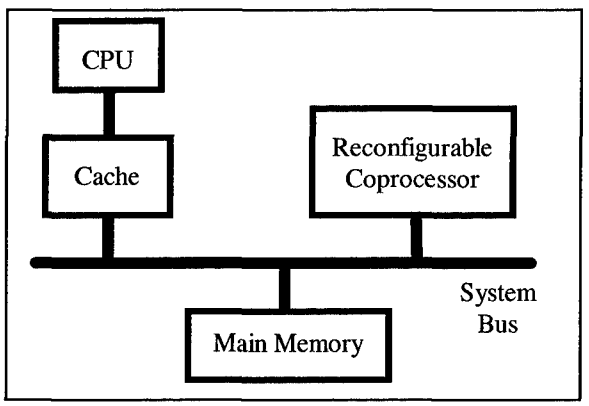

Figure 3: Coprocessor Model

Communication between the processor and the reconfigurable coprocessor is done over the system bus, and is therefore limited by bus speed. The bus can also adversely affect reconfiguration time, in the cases where the bus cannot provide the configuration bits to the FPGAs as fast the device can accept them. In this case, bus speed becomes a limiting factor. Bus speed becomes very important during execution, as it determines how fast the processor and the reconfigurable system can communicate. The bus width limits the amount of data that can be transferred between the two devices in parallel.

2.3.2.2 The CHAMP I System

An example of a loosely-couple coprocessor system is the Configurable Hardware Array Mappable Processor (CHAMP I), developed by Lockheed Sanders for the Air Force's Wright Laboratory [11]. Wright Laboratory has been working to determine whether reconfigurable architectures can provide performance near the level of ASIC

hardware, while maintaining the flexibility of software-based approaches. As an example application, they chose to implement an Infrared Missile Warning System (IRMW). The CHAMP system was developed to investigate the advantages of such a system. Based upon a coprocessor model, the reconfigurable logic is connected to a conventional workstation by the VME Bus. The CHAMP I system is constructed from 8 Processing Elements (PEs), each of which contains 2 Xilinx XC4013 FPGAs. The PEs are connected in a ring and supplemented by a crossbar connection network. In addition, a 16K by 32 bit dual port RAM is provided on board. All told, the system can implement roughly 350,000 gates of digital logic [11:8]. The CHAMP I system architecture is shown in the following figure.

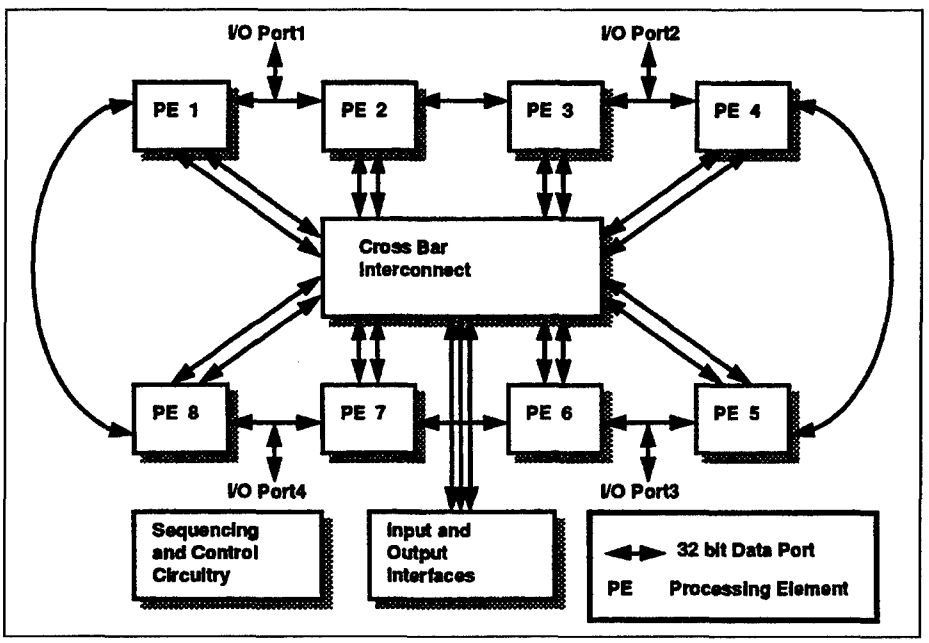

Figure 4: CHAMP I Architecture

The IRMW algorithm was originally implemented in the C programming language. Engineers at Wright Laboratory manually identified and removed those

portions of the code which could be effectively implemented on the CHAMP board, From this partition, they created a hardware schematic and software code for the host processor. The process of placing the application on the hardware required several months. The hardware implementation proved to be faster than the original implementation, in the range of 4 to 25 times faster, depending on the portion of the algorithm examined. [11:15].

Communication speed proved to be a problem with the CHAMP system, as all communication takes place over a very slow VME bus [11]. Information had to be transferred in serial (one byte per bus cycle), and could not provide the high speed communication necessary to do real-time processing of data in applications in which the host system provides the data. CHAMP was capable of real-time operation when the data was provided by a custom hardware interface to an external camera. Other limitations of the system shortage of on-board RAM (since CHAMP could not access the workstation's main memory) and a limited number of CLBs [12:43].

2.3.3 The Dynamic Instruction Set Model

The final model of a reconfigurable system incorporates reconfigurable logic into the processor itself. The processor is either wholly or partially reconfigurable. This model is often referred to as the tightly-coupled model, since the reconfigurable hardware is closely integrated into the processor itself. This is the most complex of the three families of reconfigurable architecture, and was the last to be investigated. Most of the

examples of this type of architecture exist only as proposals or FPGA prototypes. No full-fledged systems exist.

Dynamic instruction set systems get their name from the fact that the instruction set is not fixed, and can be modified to provide custom instructions to the application. Whereas the coprocessor model can be seen as providing custom functions to an application that can be called by the software application, the dynamic instruction set computer can provide custom *instructions*. A certain number of the system's assembly language opcodes are assigned as dynamic instructions, whose exact functionality depends on the configuration of the FPGAs. The intent of dynamic instruction set computers is to eliminate the costly communications overhead incurred by coprocessor systems by pulling the reconfigurable logic into the processor itself.

Two subclasses of dynamic instruction set computer have been proposed. In the first, a conventional fixed processor is augmented with one or more FPGA execution units. For example, in addition to hardware fixed-point and floating-point adders, multipliers, and dividers, several FPGA units would be included in the execution stage of a pipelined processor. An FPGA block could be configured to perform the bit-reversal operation (which is very common in many DSP applications) in a single cycle. It could then be reconfigured at a later time to perform a different function. The other subclass is the completely reconfigurable processor, in which all logic is implemented on the FPGA. In this case, the entire pipeline structure of the processor could be adapted to the application.

The Dynamic Instruction Set Computer (DISC) is a completely reconfigurable processor [49]. DISC is based upon a single National Semiconductor CLAy31 FPGA, coupled to an external memory and a configuration controller FPGA. The CLAy31 is one of the few FPGAs that can be partially reconfigured. The DISC system uses this capability to implement what is called *cache logic*. Cache logic allows functional blocks to be placed onto the FPGA, where they can be used to provide custom instructions for the processor. After use, the blocks remain on the FPGA until the space is needed by other functions. If a functional block is still in this cache when a later instruction needs the same operation, there is no need to program the FPGA. The communications overhead between the conventional functional blocks and the reconfigurable (custom) blocks is very small, since they are all implemented in the same FPGA. The disadvantage to this system is that it requires a great deal of FPGA space, and is very slow. Current FPGAs cannot implement the large amounts of logic required of floating point adders, instruction and data caches, etc., on a single device. The DISC system provided only limited functionality for this reason. Full systems would require several FPGAs. Most importantly, an FPGA requires more die area and operates at a slower frequency than a fixed VLSI implementation of a given execution unit. If a functional unit is frequently used (such as a fixed-point adder), it is much more effective to implement it in fixed hardware.

It is for this reason that the other subclass of dynamic instruction set systems has been proposed. The more commonly used functional units are implemented in fixed logic, while a certain amount of reconfigurable logic is provided for the more uncommon,

application-specific functions. A proposed architecture for such a system is shown in Figure 5.

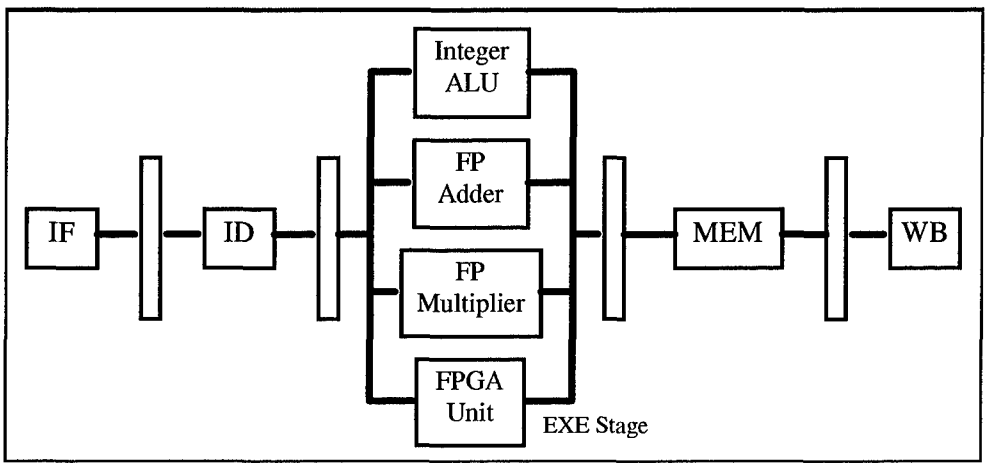

Figure 5: Typical Reconfigurable Processor Pipeline Architecture

An interesting example of this type of architecture is the OneChip [50]. This system is based upon the widely-known MIPS architecture, augmenting the standard pipeline with two Programmable Function Units (PFUs) in the execute stage. The system is currently implemented in prototype form on the Transmogrifier-1 field programmable system, but a custom VLSI implementation is planned. The specified uses for the system are for application specific hardware accelerators, and for glue logic for use in embedded controller applications. The PFUs are configured upon system (or program) startup, and are not reconfigured during operation. The OneChip has been tested on several applications, and shows promise. For a discrete cosine transform application, the system achieved a speedup of 40 times over the same program running on a MIPS R4400 processor. The major weaknesses of this system are the inability to reconfigure the functional units during execution, and a lack of application development tools.

Three other examples of dynamic instruction set processors are known. Harvard's PRISC system is a fixed-core reconfigurable processor which creates reconfigurable execution unit blocks that can implement combinational logic [41]. Only those functions which can be executed in one clock cycle can be implemented. It has been clearly defined and is implemented in simulation, unlike the DPGA-coupled microprocessor from MIT [17]. This architecture exists as a proposal only, discussing the use of the DPGA-type of programmable logic devices (as discussed in Section 2.2.5). The third example is the NanoProcessor, developed at Brigham Young [48]. This architecture is similar to the PRISC, but it allows functions with latencies of more than one cycle to be implemented. However, the static control module which forms the core of the processor is implemented in the FPGA as well. No mention is made of a custom VLSI implementation of this static logic.

Of the three families of reconfigurable architecture, the dynamic instruction set computer offers the greatest potential, but at the same time the greatest challenges. Since the reconfigurable logic is incorporated into the processor itself, the communication bottleneck which causes problems for the other two families is significantly reduced. The reconfigurable logic can be used to greatest advantage. Unfortunately, this model is the most different from present architectures. The problems involved in creating such a system have never been fully investigated. In addition, non-trivial dynamic instruction set computers require custom VLSI implementations. Static logic and coprocessor systems can be built by adding FPGAs to conventional processors, at a much lower cost. Because of these difficulties, dynamic instruction set systems are only beginning to be

investigated. The exact architecture of such systems are still unclear, as are the many difficulties involved in their implementation in a larger system and their integration with software. Perhaps most important is the complete lack of software development tools. The design of a compiler for a dynamic instruction set system will undoubtedly prove a challenging task.

2.3.4 Summary

While many examples of reconfigurable systems were given in this section, many others were left out. The architectures discussed provide a broad overview of the architectures being investigated and a frame of reference for the remainder of this thesis. While the closely-couple dynamic instruction set model appears to offer the best performance and may be the model most used in the long term, the coprocessor approach is immediately realizable. It offers many of the benefits of the closely-coupled model, although it has a higher communication cost. The coprocessor model has the advantage, however, of being very similar to many embedded controller architectures, and as such can benefit from hardware/software codesign research.

2.4 Tools for Application Development

Perhaps the biggest difficulty to overcome in reconfigurable computing is the lack of integrated development tools. The task of simultaneously developing an application in both hardware and software is very difficult. There are very good compilers, debuggers, and other tools for the development of software-only applications. Likewise, there are tools to synthesize, simulate, and debug digital hardware. Unfortunately, reconfigurable

applications consist of both software and hardware components, which must be designed and simulated in a cooperative manner. Software compilers use a fixed hardware model, in which the assembly language (and thus the capabilities of the hardware) are set. But in reconfigurable systems, the hardware capabilities can easily be changed to better suit the application being developed. Software compilers for systems in which the hardware capabilities are not specified are the subject of research.

The two major issues in the design of combined hardware/software systems are *partitioning* and *simulation.* An application can be divided into parts, each of which must be implemented either with software or hardware. The determination of the optimal partitioning (one optimizing speed, cost, memory, or other factors) is very difficult, and often requires an iterative process in which many partitions are created and simulated. The simulation of developmental hardware and software at the same time can be quite complicated and time consuming. In most modern designs, the partitioning process is done by hand.

This section describes the most common approaches to the codesign of applications for reconfigurable systems. First, several design issues for reconfigurable applications are introduced. A few of these issues are similar to those in another area of research, hardware/software codesign. The remaining sections focus on four approaches to reconfigurable application development: schematic entry, hardware description languages, custom hardware description languages, and automated extraction and synthesis.

2.4.1 Design Process

The design of an application for a reconfigurable system involves the codesign of hardware and software. Any application can be broken into blocks, at the task, functional, basic block, statement, or operational level. The designers decide whether to implement the functionality of each block using either specialized hardware or as software code. From this partitioning, the actual hardware can be designed and the software code generated. At this point, the prototype design can be evaluated to determine whether it meets the design requirements for correctness, throughput, memory size, etc. The design process is thus broken into three distinct phases: partitioning, synthesis, and simulation.

2.4.1.1 Partitioning

An important part of the application design process is a requirements specification. Certain applications must satisfy real-time operating constraints. For other applications, minimizing the system cost is the most important factor. Other constraints include software memory requirements, throughput, and hardware area. The choice of partitions will ultimately determine whether a design meets these requirements. The proper choice of partition depends heavily on the specific requirements for the application. A particular partition may be adequate for one set of requirements, and totally inappropriate for others.

Determining the best partition for a particular application is an involved process. Since each block can be implemented in either hardware or software, an application

composed of N blocks has 2^N possible partitions. While a certain block may be implemented better in hardware when viewed in isolation, a software implementation may result in a better overall system. The problem is very similar to the classical knapsack problem (multidimensional optimization). A non-optimal local decision may be needed to find the optimal global solution. As a result, a greedy partitioner will not necessarily find the optimal partition. The partitioning algorithm must search the solution space to find the best solution. This process is made slightly easier with the knowledge that not every type of operation can be implemented efficiently or easily in hardware (such as some recursive functions, etc.) or software (A/D conversion). The sample space can thus be reduced, but there is still a large set of possible solutions that must be tested to find the best one.

To determine the value of a particular partition, estimates are made of the important characteristics of the partition. Estimates are made of software, hardware, and overall run-times, and of hardware cost, memory size, etc. Partitions which do not meet the specified requirements are discarded. Ultimately, a partition is found which meets the requirements. A non-optimal partition that meets the requirements is often chosen to shorten search time. This is an important point to consider since most of this estimation and partitioning is currently done by hand. Research into automating the partitioning process for embedded and reconfigurable systems is discussed in more detail in Section 2.4.5.

2.4.1.2 Synthesis

Once a partitioning of the algorithm has been achieved, the design must be created. The synthesis phase involves the actual design of the hardware and software. From the specifications developed in the partitioning phase, hardware can be designed. In the same manner, the software code can be written. In the conventional design process, the two design teams can proceed independently of each other. The hardware designers know what capabilities the hardware must provide, while the software developers know what the hardware platform looks like and what is left to implement in software.

There are many choices of design tools for both hardware and software. Hardware developers can use schematic entry tools such as Synopsys, Cadence, Mentor, etc. (See Section 2.4.2) They may also choose to synthesize the design from a hardware description language such as VHDL or Verilog (See Section 2.4.3) It is also possible to develop the hardware from a custom hardware description language more closely resembling conventional high level languages (See Section 2.4.4). The problem is more difficult for software developers. Current compilers create code for processors with fixed capabilities. Since they attempt to optimize code for a fixed architecture, these compilers are ill-suited to take advantage of a system which can modify itself to better suit the software. For this reason, software development usually proceeds after the hardware architecture is completely specified, often resulting in a poor partitioning of the algorithm.

2.4.1.3 Simulation

Once the design is created, it must be tested to verify that it meets specified requirements. There are two alternatives for the testing of a design: prototyping and cosimulation. For small designs, it may be possible to construct actual prototypes to test the application. For larger designs, however, cost is prohibitive, and computer simulation of the hardware and software is used. This is particularly important when multiple iterations of the design process will be used to find an optimal design. The estimates of performance made in the partitioning stage can be verified or refined by simulation. If the design is unsatisfactory, the simulation information can be used to re-partition the algorithm and produce a partitioning with better performance.

Tools exist to simulate both digital and analog hardware. Digital logic simulators typically simulate VHDL or Verilog designs. Analog simulators include packages such as SPICE. While these simulators typically do a good job in simulating the functional and timing characteristics of the hardware, they run very slowly. Hardware simulators run at a small fraction of the speed of the real system. This has a serious impact on the simulation of the software system.

The major challenge of simulation of reconfigurable applications is the simulation of the combined system. This simulation involves testing software on a hardware platform that exists only as a simulation. Since hardware simulation is slow even for small test programs, simulation of a non-trivial software program can be extremely time consuming. In many reconfigurable systems, most of the software runs on a conventional

processor, and can be easily simulated at high speeds. The interfaces to the hardware can be replaced with stubs which simulate the operation of the hardware components. This provides a method of simulating part of the software, but the interface between hardware and software remains untested.

An example of a cosimulator was developed at Carnegie-Mellon [44]. The cosimulator was part of a larger codesign system, targeted toward a system architecture in which a general purpose processor is augmented by application specific hardware. A Verilog logic simulator is used to simulate the behavioral description of the hardware partition. The software portion of the system is compiled and executed on the actual processor. The major difficulty in simulation is simulating the interface between them. Carnegie-Mellon's system allows the Verilog simulator and the HLL program to communicate directly, through UNIX sockets. The intent is to allow the Verilog simulation to act as a drop-in replacement for the actual hardware. While compromises had to be made, it illustrated what would be required of a cosimulator for codesign (and also reconfigurable) systems.

The goal for cosimulation is an environment which can simulate both the software and hardware components of a system together, with a minimum of modification to the code. In addition, designers would like this simulation to execute as quickly as possible, both to speed the verification of a model, and to acquire partition profiling information for use in later partitioning steps. There are good software tools to simulate both software

and hardware independently. Researchers in this area do not seek to create new tools so much as to create better interfaces between them.

2.4.1.4 Hardware/Software Codesign

Hardware/software codesign is a design philosophy espousing communication and cooperation throughout the design process. It finds its roots in the development of embedded systems, although many of its techniques can be applied to other areas, including reconfigurable systems. Codesign aims to overcome some of the disadvantages of the conventional approach to system design through better communication between the software and hardware development paths. In this manner, any problems can be found and overcome much earlier in the process, at a substantial savings in time and cost.

Codesign tends to focus on the partitioning and cosimulation problems discussed earlier. Since design modifications become increasingly more difficult as development progresses, it is very beneficial to find a suitable partition (and thus hardware and software specifications) early. It is better to spend more time doing the initial partitioning and investigating more alternatives than it is to create an entire design and then have to redesign major portions of it when it does not meet specifications. The focus on communication and cosimulation is intended to allow earlier simulation of the system design, so that problems may be found and corrected as early as possible. A good introduction to hardware/software codesign may be found in [42].

Hardware/software codesign is in some ways very similar to the design of reconfigurable applications. In both cases, the details of the hardware are initially flexible, and it is up to the developer to partition the application between hardware and software. Much of the information about partitioning that will be presented in the following sections is intended for use in codesign systems. The major difference lies in the nature of the hardware platform. Most codesign systems assume that once the partition is achieved, the hardware design will be implemented in fixed hardware. This hardware is usually created with ASICs, although FPGAs are increasingly used as well.

In any case, the hardware is not intended to be modified by the application or by other applications, as is often the case with reconfigurable systems. For example, FPGA configuration does not have to be performed, and is thus not relevant to static codesign partitioning. If the FPGAs in a reconfigurable system are reconfigured during the operation of the program, configuration time is often a key issue in overall performance estimation. In addition, although there may be a limitation on the amount of hardware used by a design, the nature of the limitation is different for reconfigurable systems. Here, the limitation in hardware is only instantaneous, not absolute. FPGAs can implement a certain amount of logic at any time, but can be reprogrammed as often as desired, providing an effectively unlimited amount of hardware. So, while there are many similarities between hardware/software codesign and reconfigurable application design, there are several unique differences which must be taken into account.

2.4.2 Schematic Entry

Until very recently, the most common approach to the design of the hardware portion of a reconfigurable system was schematic entry. Schematic entry tools can range from the logical gate level to the VLSI layout. In some packages, the configurable logic blocks of the FPGAs themselves can be directly programmed (e.g. Xilinx' XBLOX package [52]), and the routing performed manually. All of these systems require the developer to manually describe the hardware at a low level. In the case of the first iteration of the CHAMP system, the placement of logic onto the FPGAs is done by hand [11]. The CHAMP process took several months work on the part of several engineers. While it is possible to create very good hardware designs with schematic entry tools, the process is very involved, and does not interface well with the software development. In addition, no simulation can be done of the software portion of a reconfigurable system until the entire design is complete. To shorten the development time for the hardware, alternate methods of specifying and creating the hardware design were created, such as the hardware description language.

2.4.3 Hardware Description Languages

Hardware description languages (HDLs) are similar to high level computer programming languages. They are intended to describe, in textual format, the operation of electronic circuits. The two most widely used hardware description languages are VHDL (the Very High Speed Integrated Circuit [VHSIC] Hardware Description Language) [30] and Verilog. Using a HDL, the developer can describe a circuit either

structurally or behaviorally. The behavioral description is very much like a high level programming language, describing the *behavior* of the system using variables and operations, leaving the structure of the circuit unspecified. For this reason, it is easier to create than a structural design. From the behavioral description, Boolean logic can be created and minimized automatically.

HDLs have strengths and weaknesses when viewed as tools for reconfigurable application development. The largest advantages of the HDL are its similarity to high level programming languages and logic synthesis. Behavioral descriptions are much faster to develop than schematic designs. In addition, synthesis of the logic removes much of the possibility for error. Unfortunately, HDLs can only be used to describe the hardware portion of the application. Conventional HLL code must be used to describe the software portion. Some research has been done to investigate the use of an extended form of VHDL to describe both aspects of the design using an object-oriented approach [38]. The hardest problem, partitioning, must still be done by hand. The final problem lies in the number of developers who use HDLs. The number of engineers who use HDLs is much smaller than the number of engineers and software developers who can write code in HLLs. To make it easier for the users of HLLs to develop applications, hybrid HDLs are being investigated which share many characteristics of HLLs. The custom hardware description languages are discussed in the next section.

2.4.4 Custom Hardware Description Languages

Two methods of using high level languages for application development have been examined. The first approach is to replace VHDL or Verilog with a HDL that looks like a high level language. Such a language would be easier for conventional programmers to learn, and still retain most of the capabilities of other HDLs to concisely describe hardware. The Transmogrifier C project at the University of Toronto [21], and the Spyder project at the Swiss Federal Institute of Technology [31], adopt subsets of the C language as hardware description languages. This allows C programmers to more quickly learn to write applications for reconfigurable systems.

Custom hardware description languages represent a compromise between HDLs and HLLs, and in some ways incorporate the *disadvantages* of each along with the benefits. Current custom HDLs are subsets of a HLL, and thus cannot describe the range of operations that a full-fledged HLL can describe. At the same time, they cannot describe hardware as efficiently as a dedicated hardware description language. They are easier to learn than HDLs, but still represent another language to learn. To overcome this limitation, some researchers are investigating the use of standard HLLs for application design.

2.4.5 Automated Extraction and Synthesis

2.4.5.1 Overview

The term *automated extraction and synthesis* is used to describe efforts to automate the partitioning of applications between software and reconfigurable hardware. The design system starts with a single specification of the application, creating and evaluating different partitions until a partition is found which satisfies the design constraints. Using this partition, the system automatically synthesizes the hardware logic to be used by the design and the software code to run upon it. Ultimately, the goal is to create a development system which would automate the entire process, masking the details of the underlying reconfigurable system from the applications developer and allowing those with little or no hardware design skills to develop useful applications for reconfigurable systems.

There are several important benefits to be obtained from an automated process. The most important benefit is a much faster development process. It is very difficult to develop applications for reconfigurable systems. In most cases, the application must be partitioned manually, and the resulting hardware and software portions of the system designed separately. A good example is the Infrared Missile Warning application developed for the CHAMP I system by Wright Laboratory [11] (See Section 2.3.2.2). The algorithm was originally specified in C. Since the CHAMP I system is a loosely-coupled coprocessor system, the bulk of the algorithm was implemented in hardware, with only the calling program and communication functions implemented on the host

processor. The engineers at Wright Laboratory had to manually convert the high level language specification of the algorithm to hardware and then partition, place, and route the design onto the system's FPGAs. The modified software portion had to be written by hand as well. The entire process was very time consuming and error prone. An automated process would have allowed the design to be implemented much faster, performing the partitioning and synthesis steps with little or no human intervention.

The remainder of this chapter summarizes some of the research that has been done to automate the partitioning and extraction process. The research comes from two areas, hardware/software codesign and reconfigurable computing. While most of the partitioning research to date has been done for embedded systems design, the concepts are similar to those in reconfigurable systems. There are certain important differences, which will be examined in more detail later in this thesis. The following section introduces several systems which help the designer manually partition an application. The final section discusses several attempts to completely automate the partitioning process.

2.4.5.2 Manual Partitioning

Manual partitioning systems are intended to automate as much of the tedious and error-prone work of the design process as possible, allowing the developer to focus on the high level design process. These systems do not explore the space of possible partitions so much as allow the designer to investigate the partitions manually, choose a partition, and synthesize the hardware and software specification as efficiently as possible. Since most of the work of evaluating the partitions is left to the designer, large blocks are used

so that the number of possible partitions to test is manageable. In most manual partitioning systems, the application is partitioned at the task or function level. This section describes several manual partitioning systems.

The partitioning and cosimulation aspects of hardware/software codesign were the subjects of work at Carnegie-Mellon [44]. Thomas, Adams, and Schmit created a system to aid in the partitioning, synthesis, and simulation of embedded systems. Their system model is a loosely-coupled coprocessor system, connecting some amount of additional hardware to the system bus of a conventional processor system. The system allows specification of the application in a HLL, which is then used to synthesize the software and hardware models. Finally, cosimulation of the final hardware/software system can be performed.

The Carnegie-Mellon system requires partitioning to be done by manually. Partitioning is done by hand, at the task level. The application is broken up into several tasks, resulting in a relatively small number of possible partitions. In [44], the authors identify several of the criteria a developer would use to decide which blocks to implement in hardware. First, each task is examined to determine whether it is innately more suited to software. Tasks which are better suited for software implementation are those which interact closely with the operating system, making many calls to OS functions, or relying heavily on virtual memory [44:10]. The remaining tasks may be implemented in either hardware or software.

The Carnegie-Mellon work leaves the actual partitioning decisions to the developer. The authors suggest some of the important criteria to use in making the decisions, but have not automated the computation of the estimates or the decision-making process in any way. This leaves a good deal of work to the developer, but since there are usually only a small number of blocks, there are a relatively small number of partitions to test. The research at Carnegie-Mellon is useful, however, identifying several important factors which must be considered by any automated partitioning system. These factors are discussed and expanded in Chapter 3.

The PRISM project at Brown University is a compiler system designed to automate part of the application development process for reconfigurable systems [5][4][1]. PRISM-I is a coprocessor-based reconfigurable system (see Section 2.3.2). The system compiles a single high level specification written in C into both hardware and software specifications. The partitioning process is semi-automated, identifying likely candidates for implementation in hardware but leaving the decision to the developer [5:15]. The goal of the PRISM system is to allow the developer to create a single high level description of the design, leaving the tedious work of partitioning and synthesis to the compiler.

A PRISM application begins as a C program. The first step in compilation involves the identification of blocks which can be implemented as custom hardware. PRISM partitions at the functional level. The compiler examines each function in turn, using several criteria to determine whether a hardware implementation is possible. Since

the PRISM system is a research system, there are several key limitations on which functions can be implemented in hardware [5:17]:

- State and global variables are not allowed in a hardware candidate function. All variables must be passed as parameters. Local variables are allowed.
- The sizes of the input arguments are limited to a *total* of 32 bits. Likewise, the return values is also limited to 32 bits. This is due to the nature of the proof-of-concept system upon which the system is based.
- The exit condition of For loops must be independent of the input arguments.
- Floating-point types and operations are not supported.
- Synthesized structures are combinational logic only. Sequential logic is not implemented.
- Not all C constructs are implemented. For example, Do-While and Switch-Case constructs are not supported.

Functions which are not eliminated by the above criteria are flagged and presented to the developer, who is then asked to choose which functions to implement in hardware. In PRISM-I, no estimation of software or hardware performance is done [5:15]. The judgment of which functions would benefit the most from a hardware implementation is left entirely to the user. Once the user has identified the hardware functions, the compiler can build the specifications for the hardware and software components. The hardware is

generated, and the remaining C functions are compiled normally, after adding functions to communicate with the hardware.

The PRISM-I system is one of the first attempts to create a compiler for reconfigurable computer systems. It outlines what a compiler must do to create applications for custom computing machines. The post-partitioning steps of the PRISM compiler are the most interesting, particularly in the methods used to create the hardware description. PRISM-I does not provide support for application partitioning, however, neither automating the partitioning process nor helping the user identify more likely candidates for hardware implementation. No estimation of the relative benefit of each function is done. Partitioning is thus a hit or miss process, requiring the user to guess which functions would provide the most beneficial hardware implementation.

There has been other research in this area, although most of it is intended for embedded systems. Cosmos, developed at the National Polytechnical Institute of Grenoble, was developed primarily to aid in the design of digital signal processing systems [32]. It takes a single input in SDL (a system-level specification language), and produces C and VHDL. Ptolemy, developed at the University of California, Berkeley, uses an object-oriented approach [34]. The SDF (Synchronous Data Flow) language is used as the input language. Partitioning is done manually, but some support is provided to estimate the performance of the resulting system to ensure real-time constraints are met. Other research on the partitioning problem has been done by Srivastava and Brodersen [43], Buchenreider and Veith [42], and Barros and Rosensteil [7].

Manual partitioning systems reduce some of the work which must be done by the designers of applications for reconfigurable systems, allowing the automatic generation of the hardware and software specifications after partitioning is done. The most difficult task, partitioning, is still left to the designer. It is easy to choose a partition for the application, but it is very difficult to find the best one. Some research has been done to solve this problem as well, and is discussed in the next section.

2.4.5.3 Automated Partitioning

Researchers are working to develop completely automated partitioning systems for embedded systems design. Their goal is to completely automate the hardware/software codesign process, allowing the development of a mixed hardware-software system from a single specification. Synthesis of the hardware and communication code is done by the compiler. While the goal of these codesign systems is very similar to that of a compiler for reconfigurable architectures, there are some important differences, particularly in terms of the criteria used in the cost functions used in partitioning, and in the methods used to estimate the criteria.

Gupta and De Micheli have developed a system to aid in the development of coprocessor-based embedded systems in which a real-time throughput is the key characteristic [24][23]. Applications are initially implemented in the Hardware C language, a custom HDL which looks very much like C. The application is partitioned between hardware and software automatically. The partitioning algorithm is based upon iterative improvement [24:36]. Except for the program startup code, the entire

application is initially placed in hardware. The partitioning algorithm attempts to move blocks of the application to software while still satisfying performance constraints. After the application is partitioned, the hardware and software specifications are synthesized.

To evaluate the value of a particular partition, Gupta and De Micheli's partitioner must estimate the performance characteristics and hardware size. Since the primary criteria for this system is the satisfaction of timing constraints, the authors focus on software execution time estimation. The partitioning algorithm moves blocks to software, attempting to minimize the communication between hardware and software. At each step, the algorithm checks that the current partition can operate at the speed required by the timing constraints, that the processor can perform the software tasks fast enough, and that the system bus can communicate the required information fast enough.

Gupta and De Micheli's work was one of the first published efforts at automating the partitioning process in hardware/software codesign. Many of the ideas they discuss are also relevant to the design of applications for reconfigurable systems. They identify communication between the CPU and the custom logic as a major limitation to performance. The choice of Hardware C as the input language, however, has significant disadvantages. Since Hardware C is a hardware description language, the applications are limited to those which can be entirely implemented in hardware [19]. As with all HDLs, Hardware C requires the designer to be thoroughly familiar with digital hardware design.

Vahid, Gong, and Gajski focus on the partitioning algorithm [46]. Like Gupta and De Micheli, the primary goal is to satisfy a time constraint. However, a strong secondary goal is to minimize the amount of hardware used. Vahid, Gong, and Gajski suggest that the greedy approach used by Gupta and De Micheli is easily trapped in local minimums [46:216]. For example, if two blocks O_1 and O_2 communicate heavily, moving one or the other to software would reduce hardware cost, but may result in a violation of the performance constraint because of the increased communication. Moving both blocks may yield a valid solution, but would not be found by the iterative algorithm. To overcome this limitation, the Vahid, Gong, and Gajski suggest a binary constraint partitioning approach.

The partitioning algorithm suggested in [46] nests two search algorithms. The inner algorithm attempts to satisfy the performance criteria, while the outer algorithm minimizes hardware. As the outer loop iterates through different amounts of hardware, the inner loop attempts to find a partition that satisfies the performance constraints. In this way, a greedy algorithm can be used for the inner search algorithm while overcoming the local minimum difficulties. The outer loop acts like a binary search, narrowing the range of hardware available until a partition is found which satisfies constraints and uses the smallest amount of hardware.

Vahid, Gong, and Gajski's partitioning algorithm shows some promise. The remainder of their system is relatively straightforward. Their system uses behavioral VHDL as the input language for a coprocessor that is not reconfigurable. As such, it has

the same limitations for widespread use as Gupta and De Micheli's system. The nested search strategy can be adapted to a partitioning system for reconfigurable applications, but the system focuses on minimizing hardware to satisfy a fixed time constraint. The criteria would have to be modified for use in a system in which a minimization of execution time was the primary constraint. The next system looks at the problem from this point of view.

Jantsch, Ellervee, et. al., are working on a compiler which partitions a C program between hardware and software for a static logic coprocessor-based system [33]. The partitioning algorithm is based upon a dynamic programming solution to the famous knapsack problem. The partitioning algorithm examines the code at the function and loop level, attempting to move blocks to hardware to provide the highest overall execution speed.

Jantsch and Ellervee's algorithm attempts to find the partition with the highest total speedup that will fit in the available hardware. Each candidate block must have a speedup over the software implementation. Each block thus has a value, in terms of the speedup of that block, and a weight, in terms of the number of gates required to implement it in hardware [33:228]. The dynamic programming solution to the 0-1 knapsack problem is thus applied to choose the blocks to implement in hardware.

The approach used by Jantsch and Ellervee has several advantages, and some limitations. Their compiler allows the program to be written in standard C, and the

partitioner works automatically, which makes the system available to a large audience. The choice of partitioning algorithm may not be ideal, however. Their partitioning algorithm finds the partition with the largest sum total of local speedups. This may not yield the fastest overall program. For example, a block may have a large speedup over its software implementation, but represent only a small fraction of overall execution time. A different block with a smaller local speedup that represents a larger percentage of total execution may be a better choice for hardware implementation, particularly if it requires less hardware. Jantsch and Ellervee's system is aimed at static logic systems, and does not allow the hardware to be reconfigured, making it unable to take advantage of runtime reconfigurability. The amount of logic that can be implemented in hardware is thus significantly less than could be implemented, if reconfiguration was possible.

Perhaps the most applicable research, however, is the COSYMA project at the University of Braunschweig [19][8][26][27][28][9][54]. COSYMA was designed to aid in the development of small embedded real-time systems. As with the other embedded systems development tools discussed earlier, the primary goal of COSYMA's partitioning process is to minimize the amount of hardware required to satisfy some real-time operating constraint. The system is mostly automated, and partitions the input between hardware and software at the basic block or functional level. The COSYMA architecture is based upon the static logic coprocessor model. Details on the hardware may be found in [8]. The compiler is the most interesting part of the system, however, and comes the closer than any other current research to a useful compiler for reconfigurable applications.

The developers of COSYMA have created C^X to use as their input language [19]. C^X is a superset of C, augmented with constructs to allow for the modeling of timing constraints, tasks, and task communication [19:66]. This specification is partitioned, and the resulting software and hardware specifications are output as C and Hardware C, which can be used to compile the software code and synthesize the logic for the hardware respectively. A diagram of the COSYMA development process is shown in Figure 6.

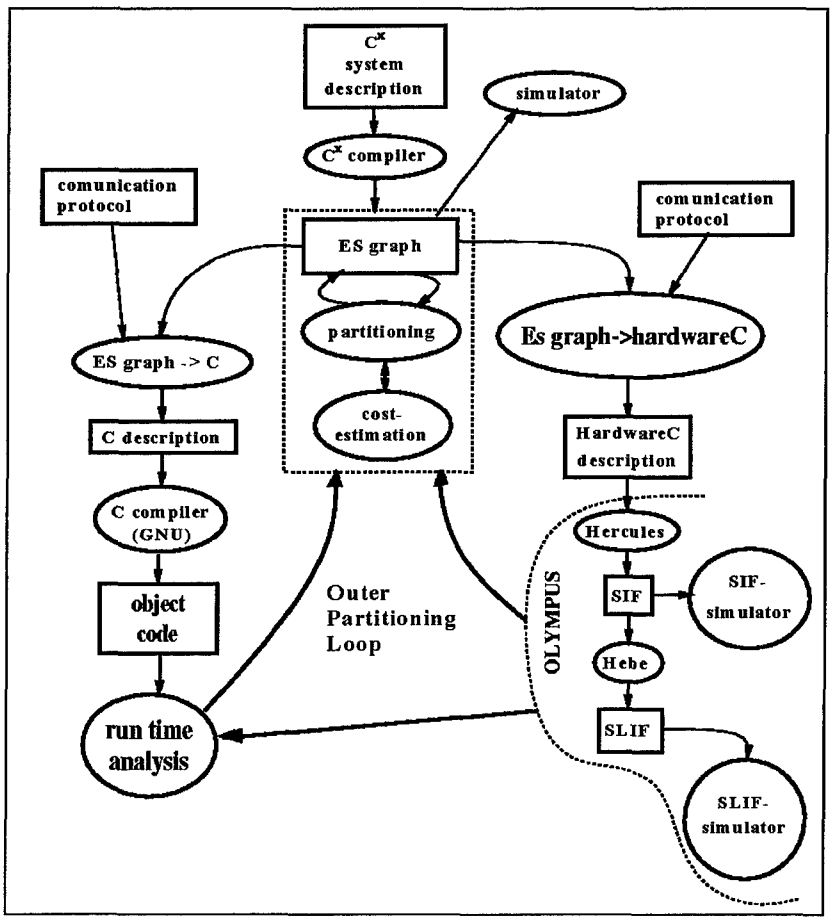

Figure 6: COSYMA Design Process

The partitioning process in COSYMA is based upon a nested partitioning algorithm similar to that proposed by Vahid, Gong, and Gajski. COSYMA uses the inner partitioning phase to perform partitioning based upon estimates of software and hardware

runtime performance, communication time, and hardware cost [46]. The outer loop is used to adjust these estimates based upon results obtained by actually synthesizing and simulating the partition obtained during the previous iteration. In this way, the estimates of the partitioning algorithm can be made more accurate as the partitioning process progresses [28]. The partitioning process begins with the entire design placed in software, and moves blocks to hardware only when real-time constraints cannot be met. As such, overall program speedup is a third priority, after the satisfaction of time constraints and minimum hardware size.

The inner algorithm of the partitioning system is based on simulated annealing [19:69]. It finds the minimum hardware partition which it estimates can meet the time constraint, and produces hardware and software specifications. The design is then simulated to verify that it meets the time constraints. The outer loop determines the difference between the estimated and actual performance characteristics; when the difference falls below a certain threshold, the current partition is accepted [28:3].

The COSYMA system is the best attempt to develop a high level language compiler for embedded systems development to date. It is automated, and its input language is very similar to conventional C. The researchers at TU Braunschweig have implemented several good techniques for the estimation of the performance and size characteristics of possible partitions with a reasonable amount of computation time. In addition, the partitioning algorithm itself seems to be effective, since the simulated annealing algorithm used in the inner phase can avoid the local minimums mentioned

earlier, and the outer phase allows the estimates to be refined with actual data obtained after synthesizing partitions.

COSYMA has several limitations, however, particularly if it was adapted for the creation of applications for run-time reconfigurable systems. The most important limitation derives from its focus. Systems developed with COSYMA are not intended to be reconfigured, and thus configuration time is not considered during partitioning. In addition, the methods of estimating hardware runtime and size are designed for ASIC implementations, where performance and area can be much more accurately estimated than they can be in FPGAs. Finally, the method of software runtime estimation is much too time consuming, since it is performed with a simulator. It is very difficult and time consuming to profile large programs with the COSYMA system. And it is those very programs which stand to benefit most from implementation in reconfigurable systems.

2.4.5.4 Writable Microcode

A final area of research concerns the writable control store. Processors using microcode control techniques can implement a portion of their microcode in writable memory. This allows a program to create new instructions by changing the manner in which the functional units of the processor are connected. Large volumes of code can often be implemented as a single complex instruction, better utilizing the fixed hardware in the processor. Techniques for the identification and implementation of software code as microcode instructions are discussed in [37].

The problem in writable microcode is similar in some ways to the problem of application development for reconfigurable machines. Computationally intensive code must be identified, and the development system must determine which blocks of code would yield the largest improvement when implemented as microcode. There are several important differences. The most important is that writable microcode changes the connections between fixed hardware units, whereas reconfigurable hardware can create arbitrary hardware. Another difference is that the overhead factors are different. Hardware cost is not a factor in microcode, for example, whereas microcode length is useful. The problems are similar in some respects, but reconfigurable computing has several unique difficulties which require unique solutions.

2.5 Summary

The task of creating a compiler for reconfigurable systems requires the combination of research from several different fields. The majority of research in the area of reconfigurable computer architectures has focused on the design of the hardware itself, and not on the process of developing applications for it. Most of the applications developed to show the merits of these architectures were created by hand, requiring large amounts of effort from skilled designers. While software tools are often used to aid in the design of hardware portion, the partitioning process until now been done by hand.

Part of the answer to this partitioning problem can be found in hardware/software codesign and writable microcode. Research in both fields has some application to reconfigurable systems, although microcode is less closely related. While the focus of

codesign research is typically on the development of small embedded systems whose primary requirements are real-time performance and small hardware size, many of the concepts and techniques of codesign can be applied to reconfigurable systems. In particular, the partitioning process can be adapted for reconfigurable systems. However, several of the estimates used for the codesign of systems intended for ASIC implementation must be changed to account for the unique characteristics of the programmable logic devices typically used in reconfigurable systems. And as shown earlier, new criteria such as configuration time must be included in the cost functions.

While many of the techniques necessary to create a compiler for reconfigurable computers have been developed for other purposes, such a compiler has never been built. Such a compiler would adapt the partitioning and performance estimation techniques used in codesign to the unique requirements of runtime reconfigurable systems. While the process is very similar in each case, the criteria used and the final goal are very different. The specifics of a compiler for reconfigurable systems are discussed in the next chapter.

III. Framework for a Reconfigurable Compiler

3.1 Introduction

In the last chapter, the need was shown for a system to automate the development of applications for reconfigurable computers. This chapter creates the framework for such a development system, called a reconfigurable compiler. The system is based loosely on a HLL compiler, although there are many important differences. Section 3.2 outlines the steps involved in this system, as well as briefly discussing the choices of input language and hardware model. The remainder of the chapter concentrates on the major tasks involved in the system: block selection, estimation, partitioning, and synthesis.

The examination of each task serves several purposes. The difficulties inherent in each task are identified, and methods for the accomplishment of each task are proposed. In certain instances, research in other areas, such as HW/SW codesign, compiler design, and hardware design can be adapted for reconfigurable compilers. Whenever possible, the techniques developed in other areas are incorporated into this system. For some tasks, particularly the estimation of hardware performance and cost, existing techniques are insufficient. The architectures used in FPGAs make it difficult to accurately estimate performance, cost, and even feasibility of a design. Accurate estimates of hardware and software characteristics are essential to the partitioning process, and the problems involved in making the estimates are examined in detail.

3.2 Overview of the Development System

3.2.1 Choice of Input Language

The development system described in this thesis uses C as the input specification language. As discussed in Chapter 2, automated partitioning research for hardware/software codesign has been done using a variety of languages. The most common languages used are high level programming languages such as C and C++. Some researchers have investigated using HDLs such as VHDL or Hardware C [23], while others have used specialized formal specification languages, such as the Specification Description Language (SDL) [32], or object-oriented specification languages such as OOFS [51]. There is no ideal choice of language for all situations. HLLs cannot always succinctly describe what would be simple hardware operations, while HDLs cannot describe some software operations. Formal specification languages are slightly better, but require the use of a specialized language. There are large amounts of C code available which may benefit from implementation on a reconfigurable system. The use of C as the input language allows the widest possible application of the system, at the expense of performance and hardware cost. These effects are discussed later in this chapter.

3.2.2 Hardware Model

The development system proposed in this chapter assumes a loosely-coupled coprocessor model as the target architecture. While less complex and expensive to implement than the dynamic instruction set model, it possesses many of the benefits. A

typical target architecture would connect a standard microprocessor and a reconfigurable coprocessor via the system bus. The coprocessor is composed of one or more FPGAs, plus some amount of additional glue logic, bus interface logic, and local memory.

For the most part, the concepts described in this chapter are independent of the actual hardware used. Two exceptions are in the estimation of performance and cost characteristics for code which may be implemented in hardware. These characteristics depend heavily on the capabilities of the logic blocks and routing resources of the specific FPGA architectures which make up the coprocessor. The examples in these sections assume an FPGA architecture similar to Xilinx' FPGAs. Xilinx is the largest manufacturer of FPGAs, and their architecture is quite typical.

3.2.3 An Example

Application development for a reconfigurable system is a complicated task. To help illustrate the process involved, this section shows how an example application can be partitioned and synthesized for a typical reconfigurable system. This section provides a high level overview of the major tasks involved.

A simple C program is shown in Figure 7. The program reads a hexadecimal number from the user, reverses the order of the bits, and outputs the new number. For example, the bit reversal of the binary number 10110 is 01101. Bit reversal is a common operation in many signal processing applications, including the Fast Fourier Transform. The operation has a very simple hardware implementation. For a 32 bit number (with bits

numbered 0-31), each bit *i* of the input is connected to bit (31 - *i*) of the output. The operation can be performed in hardware in a single clock cycle. Unfortunately, high level languages do not typically provide an efficient method of accessing and manipulating individual bits. As such, the bit reversal function is implemented as a loop, using shift, AND, and OR operations to build the output number one bit at a time.

```
#include <stdio.h>
#include <stdlib.h>

int main()
{

  char Input[10];
  unsigned int Starting;
  int Loop, Pass;
  unsigned int Building = 0;

  Pass = 34;
  printf("Please input an unsigned hexadecimal number\n");
  gets(Input);
  sscanf(Input, "%X", &Starting);
  printf("The starting number is %X\n", Starting);

     /* This is the actual Bit Reversal Loop */
  for (Loop = 0; Loop < 32; Loop++)
  {
     Building <<= 1;
     Building = (Starting & 0x1) | Building;
     Starting >>= 1;
  }

  printf("Reversed number is %X\n", Building);
  printf("Pass through number is %d\n", Pass);
  return(0);
}
```

Figure 7: C Source for a Simple Bit Reversal Program

The optimized SPARC assembly language code for the loop is shown in Figure 8. Each iteration of the loop executes eight instructions (there is a one instruction branch delay in the SPARC architecture, so the store instruction after the *bpos* branch is always

executed. With 32 iterations of the loop, a total of 256 instructions must be executed to perform the bit reversal operation.

```
L5:
    ld [%fp-36],%o1
    sll %i0,1,%i0
    addcc %o2,-1,%o2
    and %o1,1,%o0
    or %i0,%o0,%i0
    srl %o1,1,%o1
    bpos L5
    st %o1,[%fp-36]
```

Figure 8: Optimized Assembly Code for Bit Reversal Loop

The purpose of the reconfigurable compiler is to implement part of the application as custom hardware on the configurable coprocessor, so that the result has a shorter total runtime than it would if implemented as purely software code. With this in mind, the system must identify those blocks of code which have faster hardware implementations, and then decide which ones to implement given the limited resources of the hardware. For each block of code, the system determines whether to implement it with custom hardware or as software code.

Compilation begins with syntax and semantic analysis of the source code. These steps are identical to those performed by a standard compiler. At this point, the tasks performed by the reconfigurable compiler diverge from the standard compiler. The first major task is to decide exactly how much code is contained in the blocks used in partitioning. This task is called *block selection.* The statements in the code are grouped into blocks, each of which can then be placed in either hardware or software.

Block selection can be done at different levels of granularity, from functions to loops to individual operations. Coarser granularity yields fewer total blocks, and thus a smaller set of possible partitions. Since each block has two implementations (hardware or software), a program with N blocks has 2^N possible partitions. Thus, while finer granularity may provide a more efficient partitioning of the application, the cost of the partitioning increases exponentially. For this example, assume that block selection produces three blocks: the statements before the *for* loop, the loop, and the statements after the loop. There are thus 2^3 possible partitions.

As part of block selection, *hardware feasibility analysis* can be performed to remove from consideration those code blocks which cannot efficiently be implemented in hardware. These blocks are automatically implemented as software code, removing them from the set of blocks to partition. This can greatly reduce the computation required in later stages. For example, the hardware model may not allow the coprocessor to call software functions. This limitation removes the first and third blocks immediately, since they both have several calls to library functions which do not have hardware implementations. There are thus only 2^1 possible partitions to explore.

Having determined that the loop can be implemented as a hardware structure, the second major task is to estimate the performance gain resulting from a hardware implementation, and the amount of hardware required. This task is called *estimation.* For some blocks, runtime may be faster if implemented as software code instead of hardware. Other blocks may be too large or complex to implement with the available

hardware. Some blocks may have faster hardware implementations than software, but use hardware resources needed by other blocks which would yield a better overall runtime.

The estimator examines the *for* loop and makes estimates of the runtime of the software and hardware versions of the function, and the overall program runtime. Hardware cost is also estimated. The software runtime of the block is estimated as 256 cycles, based upon the assembly language code. The hardware runtime of 1 cycle shows that the hardware implementation, neglecting overhead, is significantly faster than the software. If the overhead to send the starting value to the hardware and retrieve the result is estimated as 16 cycles (2 bus operations in the mythical system), the operating system overhead is computed as 20 cycles, and the FPGA configuration overhead is computed as 80 cycles, then the total runtime for the hardware bit reversal operation is 1+16+20+80= 117 cycles. While the speedup is not 256, as it might be if the operation were implemented as a custom instruction on the CPU itself, it is still significant.

At this point, the third major task can be performed. A decision must be made whether to implement each candidate block as a software or a hardware function. This task is called *partitioning*. Using the performance and cost estimates, the partitioner determines which blocks should be implemented in hardware to provide the best overall performance. In the example, the bit reversal loop is the only candidate block. The partitioner determines that the total runtime is shorter if the block is implemented in hardware, and marks the loop for hardware implementation.

The fourth major task is called *synthesis*. The code blocks marked for hardware implementation must be removed from the software code, and hardware generated. In addition, an interface must be created to allow the hardware and software to communicate. In the example, the loop is removed from the software code and replaced it with a call to an operating system function. The operating system controls the coprocessor, handles the communication of parameters and results, and configures the FPGAs. After the function is executed, the results are returned to the calling program. The resulting code is shown in Figure 9.

```
#include <stdio.h>
#include <stdlib.h>

int main()
{

  char Input[10];
  unsigned int Starting = 0x12345678;
  int Loop, Pass;
  unsigned int Building = 0;

  Pass = 34;
  printf("Please input an unsigned hexadecimal number\n");
  gets(Input);
  sscanf(Input, "%X", &Starting);
  printf("The starting number is %X\n", Starting);

     /* This function replaces the bit reversal loop */
  HW_function(bit_reversal, Starting, &Building);

  printf("Reversed number is %X\n", Building);
  printf("Pass through number is %d\n", Pass);
  return(0);
}
```

Figure 9: C Source for a Simple Bit Reversal Program After Partitioning

The hardware specification of the function is created and synthesized. Interface logic to communicate with the system bus is then inserted. The resulting hardware is then partitioned, placed, and routed onto the FPGAs of the coprocessor, creating bitfiles which

can be used by the operating system to configure the FPGAs at runtime. The final application consists of an executable code module for the CPU and one or more bitfiles containing FPGA configuration data for the coprocessor.

This example is intended to provide a general overview of the process involved in creating a mixed hardware/software application for a reconfigurable system, and of the goal involved. There are many difficulties involved in the automation of the process, particularly as the applications get larger than the small program used here. Methods to accurately estimate the characteristics used by the partitioner are complex, as are the algorithms used by the partitioner to search the partition space to find the fastest implementation. These issues are discussed in more detail throughout this chapter.

3.2.4 Stages in the Development System

Having informally illustrated the process involved, a formal model of the reconfigurable compiler can now be developed. A reconfigurable compiler must partition an application written in a single HLL specification between hardware and software. Two types of output are created, object code for the CPU and the HW specification for those parts of the application placed in hardware. In some ways, it is similar to a conventional HLL compiler. The stages in a typical HLL compiler are shown in Figure 10. The compiler tokenizes the input during lexical analysis, performs syntax and semantic checking, and generates the object code. Optimization is usually performed on the intermediate code before the final object code is produced. Execution profiling is

sometimes used to aid in this process. More information on compiler design can be found in [2].

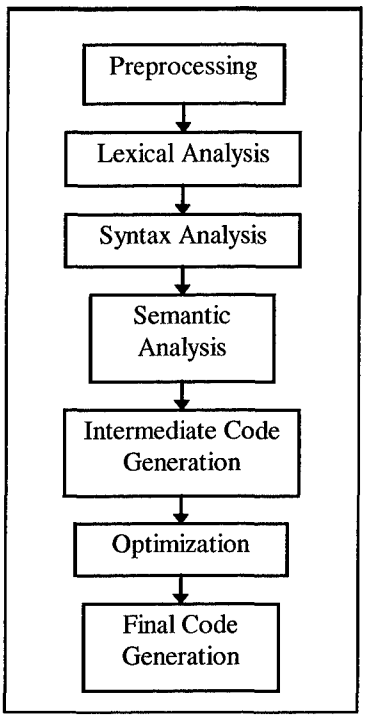

Figure 10: Stages of a Typical Compiler

The reconfigurable compiler begins in the same way as a conventional compiler. Figure 11 shows the process. C source code is preprocessed, tokenized, and syntax and semantic analysis is performed. At this point, the code is broken into blocks, to be individually implemented in either hardware or software. Partitioning depends on accurate estimates of hardware and software performance and cost. After estimates for runtime, cost, configuration time, etc., are made, the partitioner searches the *partition space* in search of best overall partition.

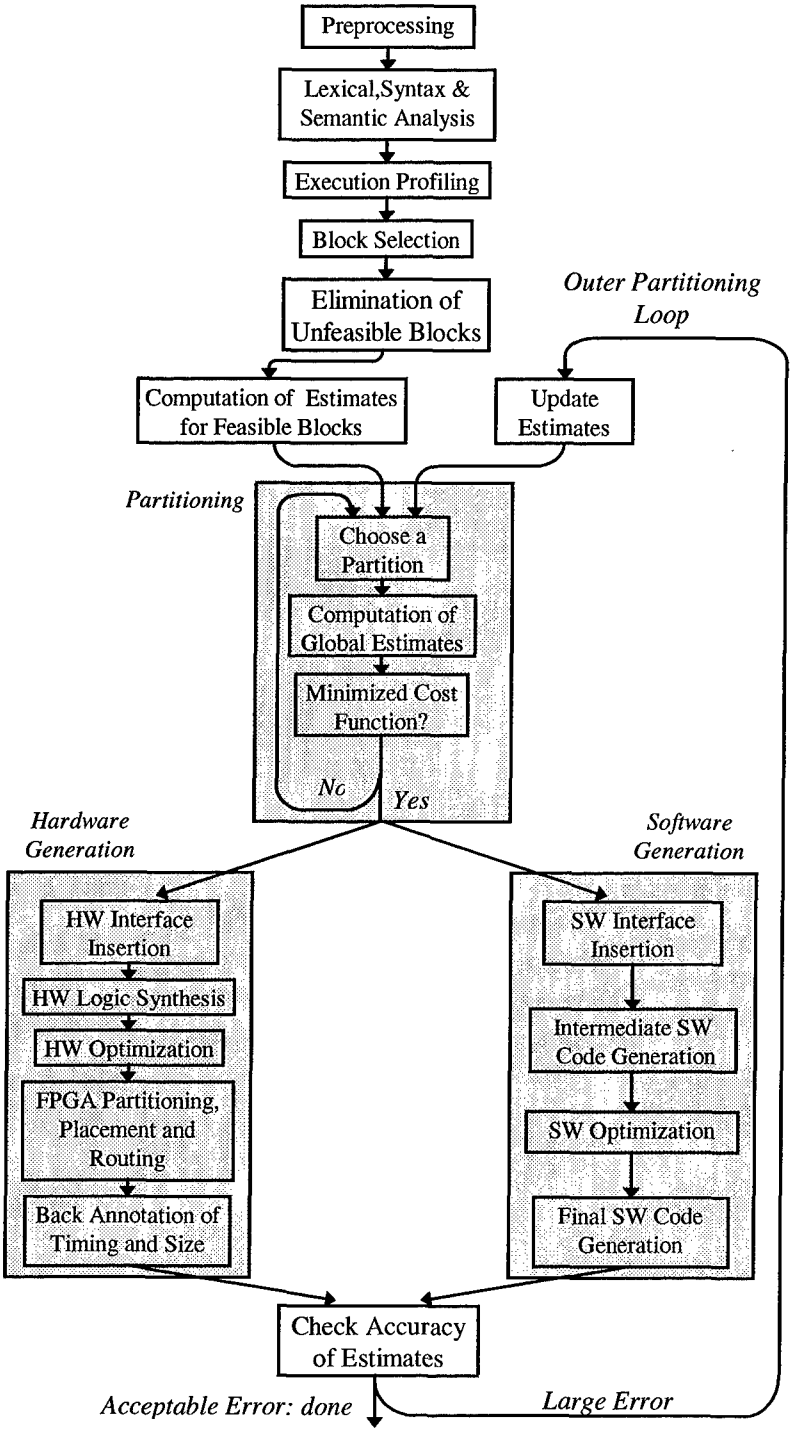

Figure 11: Stages of a Reconfigurable Compiler

Once the application has been partitioned, the hardware and software specifications can be generated. The software code is first modified to remove the code

partitioned to hardware, and code is inserted to communicate with the hardware. Section 3.6.1 illustrates how this may be accomplished. Object code is then generated as it would in a conventional compiler.

The hardware portion of the application initially exists as a behavioral specification. Interface logic is inserted to allow the hardware to be communicate with the CPU. Logic synthesis is then performed to create a structural specification of the hardware. The resulting specification can then be optimized for speed and area. The final hardware specification is partitioned, placed, and routed onto the hardware platform, and configuration bitfiles are produced. Section 3.6.2 illustrates the process in more detail.

Once routing has been performed, the actual timing characteristics and resource requirements can be determined. Back annotation of the hardware design is performed to verify that the performance of the partitioned application meets any specified criteria and that the partitioning estimates were accurate. Since the partitioner used the estimates to determine the best partitioning of the code, if the estimates were inaccurate, it is likely that the partitioner produced a suboptimal partition. The simulation results can be used to refine the estimates used for later iterations of the partitioning algorithm and increase the effectiveness of the partitioner. The partitioning, synthesis, and back annotation process can be repeated until the performance estimates approach the simulated results within a specified tolerance. This tolerance value should be a user-definable parameter, allowing the designer to determine the amount of time to spend on the process. For many applications, the time required to synthesize and back annotate multiple partitions is

prohibitive, and thus the first result would be accepted. If the error is suitably small, the partitioner's choice of the best partition should be accurate, and the final software code and hardware design can be generated.

This section has created the framework for a reconfigurable compiler. While the initial stages of this system are similar to those in a conventional HLL compiler, there are many additional stages which add to the complexity of the overall process. The following sections examine in more detail the issues involved in the block selection, estimation, partitioning, and synthesis stages of the system. Partitioning is discussed before block selection and estimation, since these tasks directly support partitioning. Section 3.3 examines the partitioning problem, and the information necessary to perform partitioning. Once these requirements are specified, the role of block selection and estimation in the partitioning process are examined. Finally, Section 3.6 examines how the partitioned code can be synthesized to create the hardware and software specifications.

3.3 Partitioning

The purpose of partitioning in a reconfigurable compiler is to determine which parts of the source program should be implemented as custom hardware and which parts should be executed as software on the CPU. The source program is decomposed into blocks, each of which can be implemented as either hardware or software. This section examines the partitioning process. Section 3.3.1 details the partitioning problem, including the primary estimates needed to evaluate each partition. Methods for the estimation of the partition characteristics are detailed in Section 3.3.2. Other factors

affecting the partitioning decision are discussed in Section 3.3.3. Finally, several algorithms used for partitioning in conventional hardware/software codesign of embedded controller can be adapted for use with reconfigurable architectures. These are detailed in Section 3.3.4.

3.3.1 Overview

For a typical reconfigurable system, the primary goal of partitioning is to minimize the overall runtime of the application. The source application is broken into blocks, and the partitioner decides whether each block should be implemented as a hardware function or left as software code. This information is then passed to the synthesis stage, which removes the hardware-mapped code and creates the hardware bitfiles and software executables.

The partitioning process can be defined mathematically as follows. Given an application composed of a set of code blocks $B = \{b_1, b_2, b_3, ...\}$, the hardware/software partition is defined as the two sets $H = \{h_1, h_2, h_3, ...\}$ and $S = \{s_1, s_2, s_3, ...\}$. $H \subseteq B$, $S \subseteq B$, $H \cap S = \varnothing$, $H \cup S = B$. The total runtime of the application partitioned into (H,S) can be defined as follows:

$$t_{run}(H,S) = \sum_{h \in H} t_{HW}(h) + \sum_{s \in S} t_{SW}(s) \qquad (1)$$

where $t_{HW}(h)$ is the runtime of block *h* as implemented in hardware, and $t_{SW}(s)$ is the runtime of block *s* as implemented as software code.

The secondary goal of partitioning is to minimize hardware cost, or size. Hardware resources are finite, and so only a certain amount of hardware can be implemented. FPGAs are still quite limited in the amount of logic they can implement at any given instant. FPGAs partially overcome this limitation since they can reconfigured during execution to implement additional hardware blocks. FPGAs are composed of three major components: logic blocks, I/O blocks, and routing resources. The amount of each type is finite, and so the partitioner must verify that each subject partition does not exceed the available resources in any of the three areas.

Hardware cost is thus composed of three expressions. The number of configurable logic blocks used by code block *h* at time *t* is $c_{CLB}(t,h)$. The number of I/O blocks used by code block *h* at time *t* is $c_{IO}(t,h)$. The measure of routing resources used by the block *h* at time *t* is $c_R(t,h)$. $c_{CLB}(t,h) = c_{IO}(t,h) = c_R(t,h) = 0$ if *h* is not present in the hardware at time *t*. The total hardware costs at time *t* are denoted as the sum of hardware costs of the individual blocks present:

$$C_{CLB}(t) = \sum_{h \in H} c_{CLB}(t,h) \tag{2}$$

$$C_{IO}(t) = \sum_{h \in H} c_{IO}(t,h) \tag{3}$$

$$C_R(t) = \sum_{h \in H} c_R(t,h) \tag{4}$$

The maximum amounts of hardware resources available at any instant are c_{CLB_max}, c_{IO_max}, and c_{R_max}. At any instant, the sum of the utilized hardware must be less than or equal to the total available hardware:

$$\forall t \geq 0, (C_{CLB}(t) \leq c_{CLB\_\max}, C_{IO}(t) \leq c_{IO\_\max}, C_R(t) \leq c_{R\_\max}) \tag{5}$$

The goal of the partitioning algorithm is to find the partition (H,S) with the minimum value for equation (1), which fits into the available hardware. The formula is shown below.

$$(H,S) = \min_{p \in P(B)} \left(\sum_{h \in P} t_{HW}(h) + \sum_{s \in B-P} t_{SW}(s) \right) \tag{6}$$

3.3.2 Computation of Runtime and Hardware Cost

3.3.2.1 Execution Model

Calculation of the runtime of a program implemented on a reconfigurable system depends to a large extent on the method of communication between the CPU and the coprocessor. This section presents one method of implementing a block in hardware. From this model, methods of computing the total program runtime can be derived. While other systems may vary in design, the methods shown in this chapter are easily adaptable to different architectures.

The source code of an application is initially targeted for implementation entirely as software. The partitioner flags blocks of code to implement as custom hardware functions, which are removed from the source code. These blocks are replaced by operating system calls to pass the parameters to the coprocessor and initiate the proper function. As an example, consider the function in Figure 12. Assume the partitioner decides to place the entire function in hardware.

```
unsigned int do_work( unsigned int Input, unsigned int A[100])
{

  int Loop;
  unsigned int working = 0;

  for (Loop = 0; Loop < 100; Loop++)
  {
        A[ Loop] = A[ Loop] + Loop;
        working = Loop | working;
  }

  return(working);
}
```

Figure 12: Source Code for a Software Function Identified for Hardware Implementation

All function calls to do_work (x, A) are replaced by HW_function(do_work, x, A). This function is an operating system or library function which acts as the interface between the software program and the hardware. Depending on the nature of the architecture, and whether runtime reconfiguration is allowed, the overhead involved in this function could be considerable, or little more than a standard software function call.

The program initiates the hardware do_work function with a call to HW_function(). The operating system determines the hardware function to be called, and whether the hardware is currently configured to implement that function. If configuration

is done only at program startup and is not modified, then the function is always present. If the hardware is reconfigured during execution, the hardware function may have been replaced by another. In this case, the operating system configures the FPGAs to implement the target function.

At this point, the operating system notifies the hardware controller that a hardware function is being initiated. If memory-mapped I/O is used, this can be done with a bus write to the controller's address, with the number of the function to be called on the data lines. At this point, the operating system can send the data values used (which were passed as parameters to HW_function()) by writing to the controller's data address.

The next step is execution of the hardware function. Some functions can be implemented as pipelined hardware, and execution can begin before all of the parameters are sent to the hardware and written into the coprocessor's registers. In other cases, as in the bit_reversal function, execution cannot begin until all parameters are received. The processor waits until execution of the hardware function is complete, determined either by polling or interrupt. At this point, the operating system reads the results, and returns control to the software application.

3.3.2.2 Runtime Derivation

As shown in Section 3.3.1, the partitioning algorithm must calculate the expected runtime and hardware cost of candidate partitions. Runtime can be computed as the sum of several components, adding operating system, configuration, and communications

overhead to the raw hardware computation time. Hardware cost is more complicated, and certain simplifications must be made. In both cases, the estimates are computed bottom-up from estimates of the individual blocks.

The runtime estimates are the most important, since it is from these estimates that the partitioner chooses the partition to use. The purpose of the hardware cost estimate is to act as a check value to ensure that the partitioning will fit on the hardware. The runtime of block *b* when implemented as software code on the host CPU is $t_{SWRT}(b)$. If implemented as a hardware module, the hardware computation time of block *b* is denoted as $t_{HWRT}(b)$. This value refers only to the raw computation time of the hardware function. The time required to pass parameters to the hardware module and to retrieve the results is denoted as $t_{comm}(b)$. The time required to configure the FPGAs in the hardware is $t_{config}(b)$. If the block is already configured, $t_{config}(b) = 0$. Finally, t_{OS} is the time required by the operating system to startup and shutdown the hardware function call. It includes the function call overhead.

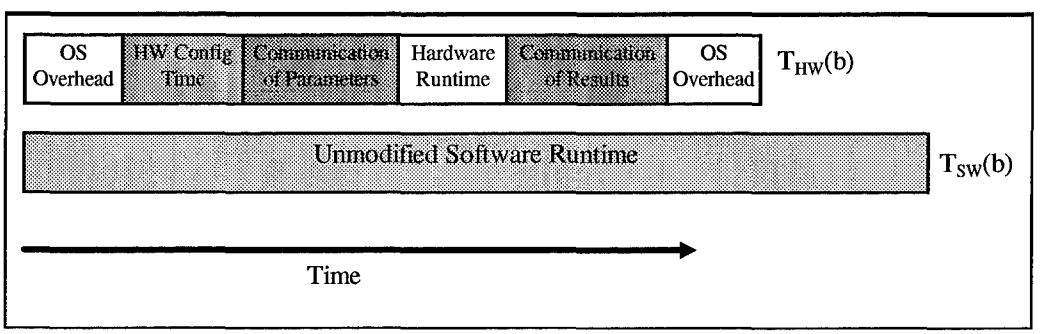

Figure 13: Comparison of t_{SW} and t_{HW} for a block b

The equation for t_{HW} for a block is thus the sum of the times to complete the events described in Section 3.3.2.1. The equation for t_{HW} is shown below, and is illustrated graphically in Figure 13.

$$t_{HW}(b) = t_{HWRT}(b) + t_{OS}(b) + t_{config}(b) + t_{comm}(b) \quad (7)$$

3.3.2.3 Hardware Cost Derivation

As used in (5), the purpose of hardware cost is to ensure that the subject partition can be implemented with the available hardware. One method of measuring hardware cost is die area, which is often used in hardware/software codesign, since ASICs are typically the target for development. Estimates of die area based upon the number of logic gates needed to implement the hardware can be quite accurate, since sea of gates implementations are often used and provide very regular structures. Estimates of the amount of communication between hardware modules are sometimes used to estimate the interconnect resources used. The combination of these two components yields a reasonable estimate of the total cost of the hardware module for an ASIC implementation.

Hardware cost is more complicated to derive for FPGAs. FPGAs are composed of three major components: logic blocks, I/O blocks, and routing resources. Internal combinational and sequential logic is implemented with logic blocks. I/O blocks are used to interface the internal logic with the outside world, in this case the system bus or other FPGAs in the coprocessor. Both types of blocks exist in finite numbers, in the tens to

thousands range, depending on the FPGA. Logic and I/O block requirements for a code block can be determined fairly accurately, as discussed in later sections of this chapter.

Routing resources are often the key factor which determines whether a design can be implemented on the FPGAs. A typical FPGA's routing is composed of connecting lines of various lengths, and switch matrices to tie them together. Unfortunately, the path of a signal through the routing cannot be predicted before routing is performed, so it is almost impossible to estimate the routing requirements of a block. Many designs which do not use all of the available logic blocks and I/O blocks on a device are unroutable. FPGA designers use a rule of thumb which specifies that no more than 75% of the specified gate capacity should be used to make successful routing probable. With this in mind, routing cost should be incorporated into the logic block cost. A scaling factor is used to overestimate the logic block cost by some factor. The FPGA is thus never filled to capacity, and the probability of routing (the real intent of hardware cost) is increased. The scaling factor is discussed in more detail in Section 3.5.3.

3.3.3 Other Factors Affecting Partitioning

3.3.3.1 Hardware Feasibility

Besides the runtime and hardware cost estimates, other factors can affect the partitioning process. One concept which can greatly reduce the search space, and thus the partitioning time, is *hardware feasibility.* Certain types of code are more difficult to implement as hardware functions than others, because of hardware costs or architectural limitations. The reconfigurable compiler may automatically target these blocks to

software, eliminating them from the set of blocks to partition. Since each block removed from the search space reduces the space by half, this can have a significant effect on the overall runtime of the partitioner.

Several types of code which may not be feasible to implement on FPGAs include:

- *Floating Point Data Types and Operations.*
- *Pointer-based Memory Accesses.*
- *Function calls to non-Hardware functions.*
- *Goto statements.*

Floating point math requires large amounts of logic. Due to the limitations of current FPGAs, 32 bit and larger floating point math cannot be implemented on a single FPGA. For this reason, most FPGA synthesis tools ignore floating point data types completely [15:11]. Implementing the operation with several FPGAs increases delay significantly, and makes it unlikely that the runtime of the hardware function will be smaller than the runtime of the software implementation.

Most loosely-coupled systems do not allow the coprocessor to access main memory directly. All data used or modified by a hardware function must be written to the coprocessor by the CPU, and read back when the function completes. For this reason, the reconfigurable compiler must determine at compile time which variables or arrays must be sent to and read from the hardware. With pointers that are modified during execution,

this is often impossible. Note that for systems which allow direct memory access by the coprocessor, pointers are easily dealt with.

Depending on the architecture used, a hardware function probably will not be able to call a software function on the host CPU. The execution model described earlier executes hardware and software in a sequential manner. The software function calls a hardware function, and stalls until the result is returned. It is possible for the coprocessor to call a software function, use its results, and finally return its own results to the operating system. For this to work correctly, the hardware must know the address of the function it wishes to call, and the operating system must know how to deal with the appropriate hardware interrupt signal. Additional overhead is needed in both the operating system and the hardware controller to handle the communication protocols. No known architecture provides this capability.

Almost every HLL text strongly discourages the use of *goto* statements, calling its use a bad programming practice, and citing negative effects on readability and on code optimizers. Additional problems are created for reconfigurable architectures. While the code in a hardware block may have several points of exit from that block, there is always a single join point in the calling code from which execution can continue. With a *goto* statement whose label is outside the hardware block, this is no longer the case. The statement after the calling function is no longer the sole point of return. This makes the interface with the operating system considerably more complex. The hardware function

would return the goto label's address upon return, at which point the operating system would change the application's program counter to the new address.

Goto statements whose target label is inside the block are also difficult. *Goto* statements have no equivalent in VHDL synthesis, making creation of the hardware specification much more complicated than it would have been had structured branch statements such as loops and switches been used instead. As will be seen in the hardware synthesis section, statements which modify the control flow of the code can have complex hardware implementations. The reconfigurable compiler could be made to handle *goto* statements of this type, at the cost of a more complicated implementation.

The choice of which constructs of the HLL to exclude from hardware implementation is up to the specific reconfigurable compiler. The reconfigurable compiler may exclude other constructs not discussed here. Based upon the gate limitations of current FPGAs, reconfigurable systems, and those codesign systems which partition C for embedded controller design, the limitations listed here would seem to be sufficient.

3.3.3.2 Interdependence of Estimates

As shown in Section 3.3.2, estimates for the runtime and size of the entire partitioned application are created from estimates for individual blocks. Unfortunately, while most of the estimates for a block can be computed in isolation, one of the estimates is dependent on the partitioning of the blocks around it. The hardware communication

time depends on which variables must be sent to hardware before execution, and retrieved afterwards. If one or more of the variables was already passed to the coprocessor for a previous hardware-mapped block, or it was defined by a previous hardware-mapped block, it does not need to be passed again. The communication time of the block can be reduced accordingly.

A more detailed discussion of this problem is included in Section 3.5.4. For now, it is sufficient to note that care must be taken when constructing the global runtime estimate from block-based estimates.

3.3.3.3 Runtime Reconfiguration

The largest single difference between reconfigurable systems and the embedded systems that are the subject of hardware/software codesign research is the incorporation of dynamic reconfiguration into the execution model. HLL-based codesign systems partition code between software and a hardware coprocessor, but configuration is done at design time and the hardware is usually implemented with ASICs. Although FPGAs are sometimes used, runtime reconfiguration is not available to the embedded system developer. On the other hand, while reconfigurable systems have a fixed amount of hardware which limits the size of a single hardware function, the hardware can be reconfigured during execution to provide an effectively unlimited amount of hardware to the application. The problem in reconfigurable systems becomes on of how best to utilize the hardware.

Unfortunately, configuration of FPGAs can be a very time consuming process. While most FPGAs can be completely configured in milliseconds, this is an extremely long time for microprocessors with clock periods measured in nanoseconds. Section 3.5.5 shows that configuration time can be very large, even for FPGAs which allow partial reconfiguration. For FPGAs which do not allow partial reconfiguration, the penalty for configuring the entire FPGA must be paid every time a hardware function must be changed on the FPGA. If several hardware functions are contained in a bitfile, it is desirable to use each of the functions at least once before the bitfile is replaced by another. Otherwise, the configuration time penalty for the entire bitfile is spread among only those function which are actually called.

For example, consider an application containing five functions which have been selected for hardware implementation: A,B,C,D, and E. The FPGAs used in the coprocessor do not allow partial reconfiguration. Two bitfiles are created arbitrarily, containing functions A,B, and C, and functions D and E, respectively. If function C is to be called, the entire bitfile containing all three functions must be loaded. If all three function A, B, and C are called before the FPGA is reconfigured, the configuration time penalty is spread between the three functions as expected. If only function C is called before the FPGA is reconfigured, the configuration time penalty for all three functions is added only to function C. The overall hardware runtime of function C suffers heavily as a result.

To reduce this problem, the partitioner should make groupings of hardware functions into bitfiles during the partitioning process. For FPGAs which do not allow partial reconfiguration, the partitioner should group functions which are likely to be called close to each other during execution, with the intent that all functions in the bitfile are used before another bitfile must be loaded. One of the goals of the partitioner is to minimize the number of times the FPGAs must be reconfigured during execution, even when execution flow does not match the results of the execution profiling.

For FPGAs which do allow partial configuration, the difficulty is reduced, but not eliminated. Hardware functions are placed and routed onto the FPGAs to fixed logic blocks. They cannot typically be moved about on the FPGA to fill whatever logic blocks are not currently being used by other functions. For example, if hardware function A is routed to the FPGA as shown in Figure 14, a second hardware function B may be routed to the same blocks. If these two functions are expected to be called together in a sequence such as ABAB, the FPGA would have to be configured 4 times.

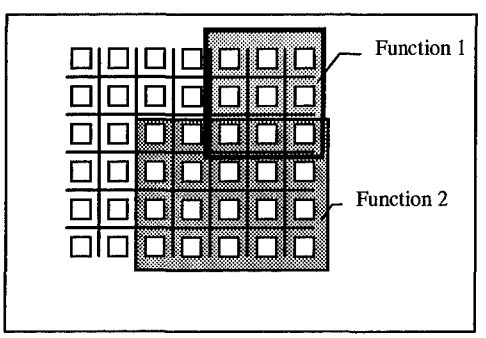

Figure 14: Two Overlapping Hardware Functions

If the two functions were associated with each other and routed together as shown in Figure 15, a single bitfile would be created for both functions. Configuration would only have to be performed once, and configuration overhead would be reduced significantly.

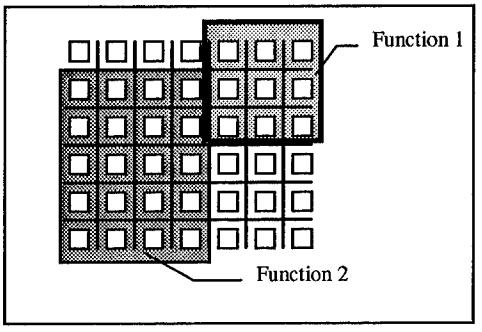

Figure 15: Non Overlapping Placement of the Two Hardware Functions

It can be seen that runtime reconfiguration offers significant advantages with the elimination of the fixed hardware limitation, but at the expense of a more complicated partitioning process. The partitioner must decide not only which code blocks should be placed in hardware, but also how to group them together so that configuration penalties are minimized and the highest overall speedup is achieved. The grouping of hardware functions into bitfiles is discussed in more detail in Section 3.5.5. Possible approaches to the partitioning problem are discussed in the next section.

3.3.4 Partitioning Algorithms

The partitioning process is based upon a nested loop, as shown in Figure 11. The inner loop is the partitioner, which decides whether to implement each code block in either hardware or software to create an application with the shortest possible runtime.

The inner partitioning algorithm operates on estimates of timing and hardware cost. The outer loop uses the results of the synthesis and routing processes to update these estimates so that the next run of the partitioner generates a more accurate result. This section explores the inner loop partitioner, including the difficulties unique to reconfigurable applications development, and possible search algorithms which may be used.

3.3.4.1 The Partitioning Decision

Partitioning is an iterative process. The algorithm will choose a partition from the partition space, and compute the overall runtime based upon (1). Multiple partitions are chosen and discarded by the inner partitioning loop until the one with the smallest overall runtime is found. The partition space is composed of all of the code blocks which were not disqualified by hardware feasibility analysis. Since each of these blocks could be implemented in either hardware or software, the search space could be extremely large. Intelligently searching the space is the goal of the partitioning algorithm.

Since the amount of hardware available in a reconfigurable system is effectively unlimited, one may be tempted to automatically implement in hardware all blocks whose hardware runtime is less than the software runtime. This would effectively make the partitioning process unnecessary. Unfortunately, the configuration time estimates depend heavily on the grouping of blocks in bitfiles and the number of times reconfiguration must be done, particularly for coprocessors composed of FPGAs which are not partially configurable. Configuration time can be a major factor in overall runtime, and it is often

the case that the best partition is one which maps only a subset of the possible blocks to hardware.

One effect of configuration times is that the partition chosen should require as few reconfigurations of the hardware as possible. This holds true even for partially configurable FPGAs, since configuration incurs operating system overhead which may be independent of the size of the block being configured. Therefore, the grouping of hardware functions into bitfiles becomes important, and must be considered by the partitioning algorithm.

To illustrate the partitioning problem, consider the following example. An application is broken into blocks, and 5 blocks are identified which may be implemented as hardware functions, A, B, C, D, and E. The remaining code is combined for the purposes of this example into a single block, F. The software and hardware runtimes, hardware cost, and local speedup are estimated for each block, as shown in Table 1.

Block	Feasible?	Calling Frequency	t_{SWRT}	t_{HWRT} (neglecting t_{config})	Local Speedup (t_{SWRT}/t_{HWRT})	Hardware Cost, c(B)
A	Y	1	8	2	4	6
B	Y	1	3	1	3	4
C	Y	2	20	10	2	4
D	Y	1	2	1	2	3
E	Y	1	3	2	1.5	2
F	N	1	100	---	1	

Table 1: Performance and Cost Estimates for Partitioning Example

Execution profiling reveals that the most likely order of calls to the hardware candidates is AECBDC. The hardware can implement 10 units of hardware at any

instant. Five different partitions are created, and the overall runtime of each is estimated, as shown in Table 2. It is assumed that partial configuration is not possible. Configuration of the entire FPGA requires 4 time units. Total runtime is computed by adding the software runtime of the code that cannot be partitioned to the hardware or software runtime of each of the other called blocks, depending on whether each is implemented as either hardware or software.

Partition	Bitfiles	Number of Configs	Runtime (w/o t_{config})	Total Runtime
1	[AB][CDE]	4	126	142
2	[AC]	1	130	134
3	[BCE]	1	133	137
4	none	0	156	156
5	[AE][BC][CD]	3	126	139

Table 2: Partitioning Estimates of Total Runtime

The contents of each bitfile is shown with brackets. The number of reconfigurations of the hardware is calculated by following the expected execution order and determining how many times the configurations must be swapped to execute all of the called functions. For example, the first partition contains two bitfiles, one of which contains functions A and B, and the other contains functions C, D and E. The profiled execution order AECBDC is executed as follows: load bitfile 1, A, load bitfile 2, EC, load bitfile 1, B, load bitfile 2, DC. Four reconfigurations of the hardware required.

Examining the estimates for overall runtime, the partitioner would choose partition 2, since it has the lowest overall runtime. Note that it does not place all blocks which have faster implementations in hardware. It is also possible that better partitions

exist. Only 5 of the 2^5 partitions were investigated. Partition 2 is simply the best choice of those 5 candidates. To be confident in its choice, the partitioner would have to examine other candidates as well.

To summarize the partitioning process, the partitioning loop is composed of three steps:

- *A partition is chosen from the Partition Space.* Hardware functions are selected
- *Hardware functions are grouped into bitfiles.* The goal of the grouping is to minimize the number of reconfigurations of the hardware needed. Hardware cost is evaluated to ensure that each grouping will fit in the available hardware.
- *The overall runtime expression is evaluated.*

This iterative process is repeated until a partition is found which minimizes the overall runtime.

The purpose of this example was to illustrate that the partition space must be searched to find the best possible solution. The solution to the problem is not as simple as placing all blocks in hardware whose $t_{HWRT} < t_{SWRT}$. The partition space must be searched as completely as possible. Given the size of the partition space, methods should be developed to remove blocks from consideration whenever possible, and to choose candidates as intelligently as possible. Methods for search space reduction are listed in

the next section. Several search algorithms which may prove useful are discussed in Section 3.3.4.3.

3.3.4.2 Search Space Reduction

Given the huge size of the partition space and the computation involved in function grouping and estimate evaluation, the partitioning algorithm should evaluate as few candidates as possible. Search time can be improved in two ways: reducing the number of candidates in the partition space, and evaluating the most likely candidates first. The first method is accomplished before partitioning is started, and the second is a part of the partitioning algorithm itself.

Since the partition space of a set of N block is 2^N candidates, every block removed from the set before partitioning begins reduces the search space by a factor of 2. This is primarily accomplished through hardware feasibility analysis, discussed in Section 3.3.3.1. The runtime estimates for each block can also be used to eliminate blocks from consideration whose hardware runtime is larger than the software runtime. In these cases, hardware implementation cannot improve performance. Since runtime estimates are likely to be inaccurate, blocks whose $t_{HWRT} > \beta\ t_{SWRT}$, where β is some constant <1, should be eliminated as well. The choice of β would be set as a user parameter.

Broadening this technique allows the removal of blocks whose performance gain is only a small percentage of overall execution time. The performance increase gained by partitioning these blocks may be considered too small to merit the additional computation

required to partition the extra blocks. The creation of another user parameter, κ_{min}, represents the minimum percentage improvement a block must achieve to be considered. $\kappa(b)$ is the percent improvement to be gained by implemented only block *b* in hardware:

$$\kappa(b) = \frac{freq \times \left(t_{SWRT}(b) - t_{HWRT}(B)\right)}{\sum_{s \in B} t_{SW}(s)} \times 100 \qquad (8)$$

where *freq* is the number of times the function is to be called during execution. If the $\kappa(b) < \kappa_{min}$, the block is discarded from consideration.

Another technique could be used before partitioning begins to identify the more likely candidates to the partitioner. The partitioning algorithm is unlikely to be able to search the entire partition space. Pre-partitioning analysis could identify blocks whose hardware implementation is likely to reduce overall runtime more than other blocks, and thus be part of the solution set H. By choosing the more likely candidates first, the partitioner is more likely to find the best possible partition in fewer iterations than would be possible from a completely random search of the space. Different criteria could be used, but local speedup or total reduction in runtime are likely be two of the best means of ordering the candidates.

3.3.4.3 Search Algorithms

A variety of algorithms exist which can be adapted for use in the partitioning problem. Unfortunately, the overall problem consists of several different dimensions, and

no single algorithm can take them all into account. The ultimate goal of any algorithm used would be to minimize overall runtime. At the same time, instantaneous hardware limits must be obeyed, and function groupings must be made to minimize the number of reconfigurations needed.

The partitioning problem for reconfigurable systems is thus very different from the partitioning problem for hardware/software codesign. Typically, codesign seeks to minimize the amount of hardware needed to satisfy some real time constraint on operation. Other researchers, particularly those involved in the COSYMA project, attempt to minimize runtime given a fixed amount of hardware in which to implement hardware functions [28]. The hardware in this case is a fixed bin, which can be filled by the partitioner. Bin packing algorithms can be applied successfully in this case.

Instead of trying to schedule and group hardware functions, the partitioning problem could be simplified. While the solution would be less accurate, conventional algorithms could be used. It has been shown that configuration time is a significant factor in the partitioning decision. If the partitioning goals are changed so that the two objectives are to minimize runtime while minimizing overall hardware cost (two conflicting goals), conventional algorithms can be applied, which seek to maximize speedup while minimizing overall hardware cost. The hardware costs of all of the blocks placed in hardware are summed to create this global value, allowing scheduling effects to be ignored. Total hardware cost is indirectly related to the number of reconfigurations needed, and the simplification of the problem is highly desirable.

If this approximation is used, several of the techniques used by codesign researchers are applicable. A weighted cost function could be created, combining runtime with hardware cost, as suggested by Vahid, Gong and Gajski in [46]. Greedy algorithms, such as those used by Gupta and De Micheli [24] should be avoided, since they can easily get caught in local minimums. The COSYMA system uses a simulated annealing algorithm [28]. The system proposed by Jantsch and Ellervee uses a bin packing algorithm to maximize the total speedup [33].

The accuracy of the partitioning algorithm is largely dependent on the criteria used in the weighting function. If total hardware cost is used, accuracy will depend on how closely the total hardware costs of the partitions correspond to overall runtime when configuration time is taken into account. The effects of configuration time are reduced for partially reconfigurable systems, but the problem can still be significant. The partitioning algorithm is one area which will benefit from further research.

3.4 Block Selection

An important decision facing the designers of a reconfigurable compiler is how to group code into the blocks used by the partitioner. Ideally, each operation would be partitioned between hardware and software independently. Unfortunately, the number of blocks becomes large, and the size of the partition space too great to search in a timely manner. Grouping operations together reduces the size of the search space, at the risk of producing a less efficient partition.

Two additional reasons for using a smaller number of blocks containing more code are communication requirements and estimate accuracy. It will be shown in the next sections that the communication of data between the CPU and the coprocessor becomes a large part of the overall runtime. To minimize this communication, larger blocks should be used. In addition, as the granularity becomes finer, hardware estimates become more and more inaccurate, as errors which would otherwise average out over a large hardware structure are viewed in isolation [45:461].

The most obvious way to group code into blocks is based upon code structures. A certain code structure, such as the function, loop, or statement, is chosen as the standard block size. In order of decreasing size, the code can be partitioned at the task, function, loop, statement, or even operation level. Task and function level partitioning were used in the early manual partitioning systems in hardware/software codesign since the small number of blocks results in a small solution space that can be realistically searched manually. Finer granularity results in more blocks to place, and thus a significantly increased workload which is only possible with automated partitioning. At the same time, finer granularity offers the possibility of better overall performance. The goal of block selection is to group code at a coarser level of granularity than operations or statements, but in a way such that the final partition produced is as efficient as what would be produced by partitioning at the finer levels of granularity.

There are several problems with selecting blocks based upon fixed code structures. The most important problem is that the presence of code in a block which is

individually unsuitable for hardware implementation is enough to invalidate the entire block. For example, in loop level partitioning, the presence of a software function call in a loop body would invalidate the entire loop. The remainder of the loop body may be a very good hardware candidate, provided that the offending code can be excluded from the block. If this is not done, many blocks of code which would result in speedup if implemented in hardware are not even considered by the partitioner.

Another problem is that the groupings of statements into groups like functions or loops may not best reflect the communication and computation requirements of the program. Ideally, the partitioner will move blocks of code to hardware which do a lot of parallel computation on a small amount of data. Statements that operate on the same data should be grouped together whenever possible, to avoid a situation in which data must be repeatedly sent back and forth between the coprocessor and the CPU.

For example, consider the code fragment in Figure 16. If loop level partitioning is used, two candidate blocks are identified. Since the statement between the two blocks is not in a loop, it is automatically placed in the software partition. If the two loops are both placed in the hardware partition, the data that the ineligible statement operates upon must be retrieved from the hardware before the statement, and returned to it afterwards. If dataflow analysis cannot identify the elements of the A array which are needed, the entire array must be sent. The entire A array is sent back and forth to the hardware twice, greatly increasing t_{HW} for these blocks, and quite possibly making hardware implementation for the blocks unprofitable.

```
For (loop = 0; loop < 1000; loop++) {          /* Block 1 */
    A[loop] = A[loop] + loop;
}
A[i+14] = A[i+160] = A[i+190] = A[i+500] + 5;  /* Ineligible statement, since not in a loop */
For (loop = 0;  loop < 1000; loop++) {         /* Block 2 */
    A[loop] = A[loop] - A[loop -1];
}
```

Figure 16: Code Fragment Showing How Intervening Code Can Result in Inefficient Grouping of Code into Blocks

If, on the other hand, the ineligible statement were to be grouped with one of the loops, the A array would only have to be sent back and forth once. If both loops are placed in hardware, the array can be sent to hardware at the beginning of block 1, and sent back at the end of block 2. Communication time is greatly reduced, and it is much more likely that the blocks will result in speedup.

The third problem with fixed sizes occurs at the functional level. The grouping of code into functions is up to the programmer, and may be entirely arbitrary. The amount of computation done on the given data may be very small, and the amounts of available parallelism limited. The communication time is more likely to be a significant percentage of the overall t_{HW} than it would be in a loop, resulting in a smaller probability for speedup. This is not always the case, but loops are more likely to have a greater *computational density* (work per unit data communicated) than functions.

3.4.1 Block Growth

3.4.1.1 The Ideal Case

Rather than using blocks based upon fixed code structures, the partitioner should use blocks of varying granularity. Blocks can begin at statement level granularity, and be

individually grown as large as desired. Typically, dependent statements would be merged into a block one at a time, with the intent of increasing the amount of work accomplished (increasing the chances of useful parallelism, and thus hardware speedup) while reducing the communication requirements. Blocks are grown until no additional statements are available whose inclusion would increase the computational density of the block.

Ideally, the block selection algorithm would group code which performed many operations in parallel on a small amount of data, without regard for the programmer's code structure. These *amorphous* regions could cross function and loop boundaries, creating blocks which result in low hardware runtimes and moderate communication costs.

Unfortunately, finding the best grouping of statements into blocks to send to the partitioner is exponentially difficult. Starting with a single statement, the problem of deciding whether to add each additional statement does not lend itself to a greedy solution, since adding one statement may lower the overall computational density, but it may enable a later statement to be added which ultimately raises the computational density. For example, consider a sequence of dependent statements A, B, and C as shown in Figure 17. The arrows denote the flow of data (the variables denoted by w, x, y, and z).

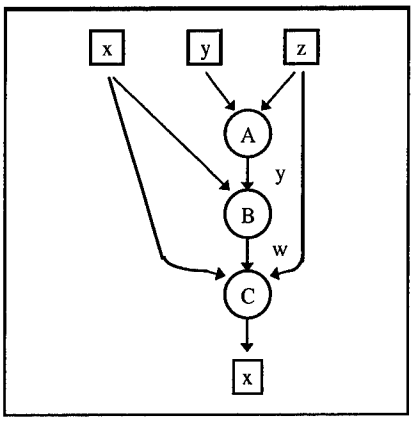

Figure 17: Example Sequence of Statements, Showing Data Dependencies Between Statements

Assume that the block selection algorithm begins by creating a block containing only statement A. Next, it must determine whether to add statement B into the block. If statement B is added, statement C may be added as well. The block selection algorithm makes an estimate of the computational density of each individual statement, and for the possible combinations, as shown in Table 3.

Block Statements	Computation	Communication	Computation Density
A	9	3 (in: *y,z* out: *y*)	3 (base)
B	1	3 (in: *x,y* out: *w*)	1/3
C	10	4 (in: *x,w,z* out: *x*)	2.5
AB	10	4 (in: *x,y,z* out: *w*)	2.5
ABC	20	4 (in: *x,y,z* out: *x*)	5

Table 3:Computational Density Estimates for Block Selection Example

The block selection algorithm quickly computes estimates of the computation and communication requirements of the statements. The computation value could be computed in several ways, from a simple measure of the amount of computation in the statement to an estimate of the speedup obtained by putting the block in hardware. The computation density of A is computed as 3 units. The final block created using A as a base must have a computation density of 3 or higher, since the statement can always be

partitioned by itself. Adding statement B yields a computation density of only 2.5, which is less effective than partitioning A independently. A greedy algorithm would stop here, since the only adjoining statement results in a decrease in effectiveness. But by adding statement C to the block, the computation density becomes 5.

Another factor that may become important is the hardware cost of the statements. By adding statements to the block, the amount of hardware required to implement the block increases, possibly to the point where it may no longer be implemented on the available hardware. Even if the block does not consume all of the available hardware resources, it may still consume a large enough portion that other blocks may not be placed on the hardware at the same time as the current block. So while adding a statement to the current block may improve the performance of the block, it may have a negative effect on the overall partition. As such, hardware cost may also be a useful factor in block selection.

In effect, the block selection problem becomes a smaller version of the partitioning problem. A search space is created, and estimates of benefits and cost are used to find beneficial groupings of code into blocks. The benefit of block selection is that the search space used to create blocks is much smaller (only dependent statements), and the estimates used can be less accurate than those used in the actual partitioning process. The estimates of computation and communication need not be as accurate as those used later, and so can be made in less time. If done correctly, the result of block

selection is a much smaller search space for the partitioner, and a shorter compilation time than would be required if each statement was partitioned individually.

3.4.1.2 A Compromise Solution

Block selection as described in the previous section is a complicated task. However, it may be possible to achieve some of the benefit of variable block sizes with much less computation. A compromise approach may be taken that begins with blocks based upon a code structure, called *seeds*, and adds or subtracts a limited number of statements to improve the probability that the block will have a beneficial hardware implementation. Loop structures are used as the starting point for the technique described in this section, referred to as *loop seeding*.

Perhaps more so than other code structures, loops are often good candidates for hardware implementation. Loops typically perform a great deal of computation, often over a small amount of data, and often possess some form of parallelism that may be taken advantage of to achieve speedup in hardware. Loops may be implemented as hardware functions in their entirety, requiring one hardware function call, or only as the body, leaving the control structure of the loop in software to call the hardware function multiple times.

Loop seeding begins by creating a block for each loop structure in the code. The statements in the loop are analyzed for hardware feasibility (as described in Section 3.4.2). If both the loop body and control expressions have hardware implementations, the

entire loop remains in the block. If the loop body is feasible, but the control expressions are not, the loop control expressions are removed from the block. If one or more of the statements in the loop body is unfeasible, the block selection algorithm shrinks the partition to contain only those data dependent statements in the body which are feasible. In effect, one or more blocks are created from the loop body, leaving the loop control structure and unfeasible body statements for software implementation.

The next stage is to try to merge loops at the loop depth (i.e. nesting level) that operate on data used by the loops. For example, the two loops in Figure 16 utilized the array A. Since the intervening statement is not in a loop, it is not considered as a candidate for hardware implementation, and greatly increased the communication requirements for the hardware implementations of the loops. If two adjoining loops are feasible, it may be beneficial to merge them into one block along with the connecting statements. In this situation, communication requirements decrease, and the hardware implementation is likely to be much faster for only a small amount of additional hardware. A separate block may be created for each loop, plus an additional block that contains both loops. The effects of this duplication of code in multiple blocks will be discussed shortly. If the combined hardware requirements are not great, combining adjoining loops greatly reduces the partition space while only slightly increasing the risk of creating a block whose hardware cost has negative effects on partitioning.

Finally, nested loops may be merged with parent loops in the same manner as with sibling loops. If the statements at the same level as the inner loop may be merged with

the inner loop with a suitably small increase in hardware cost and communication, the nested loop block may be discarded, and only the outer loop considered as a block for partitioning. As an alternative to eliminating the inner loop blocks, they may be sent to the partitioner as well, allowing the same code to be in both the inner loop block and in the outer loop block. This changes the partitioning problem slightly, since the inner loop block is a subset of the outer loop block ($B_I \in B_O$), and if the outer loop is partitioned to hardware ($B_O \in H$), then the inner loop is automatically part of the hardware set ($B_I \in H$).

The loop seeding technique described here is only presented as a compromise solution to the block selection problem. The results of partitioning are likely to be better than they would be if blocks of fixed size were used, but are unlikely to be as efficient as those obtained if each statement or operation were partitioned individually. Loop seeding is a heuristic method that uses observations of the limitations of fixed loop blocks to reduce their effects. More effective block selection strategies should be explored, however, and present several interesting possibilities for further research.

3.4.2 Hardware Feasibility

Useful for both fixed and varying block sizes, hardware feasibility analysis is a very important part of search space reduction. As stated previously, not every type of code is a suitable hardware candidate. Detecting these statements and removing the blocks containing them from the partitioning process greatly reduces partitioning time. For completely static block selection, hardware feasibility is used to find code which would invalidate the entire block (placing it automatically in the software set). Hardware

feasibility is used by loop seeding to find unfeasible statements in loop bodies, and to determine whether adjoining statements can be merged with the loop block. This section discusses feasibility analysis from the perspective of fixed block sizes, but is easily applied to other methods as well.

Feasibility analysis is straightforward. Each block is examined for code or data types which the reconfigurable compiler will not implement as hardware. Two of the four types of code identified in Section 3.3.3.1 are easily identified from an examination of the code. Blocks containing floating point data types or operations can be removed from consideration, as can those containing *goto* statements. The two remaining code types, pointer memory accesses and function calls to software functions, require more caution.

As discussed earlier, the architecture of the reconfigurable system may not allow the coprocessor to directly access main memory. Therefore, dereferences of pointers may not be allowed. Pointer dereferences are easily spotted in the code and flagged, but a complicating situation arises when arrays are considered. Repetitive and computationally intensive operations on arrays are some of the best candidates for implementation in hardware. Unfortunately, C allows the programmer to access the elements of an array in several ways, using the index operator or pointers. For example, both of the following lines of code access the same element of an array:

```
Int Iarray[100];
int *Ptr = Iarray;

Iarray[10] = 5;
*(Ptr+10) = 5;
```

Figure 18: Pointer versus Array Indexing Memory Access

This poses a problem for the reconfigurable compiler. Pointer accesses to elements in an array which has been passed as a parameter to a hardware function should be allowed, but should not be for addresses outside of the arrays. Unfortunately, it can be difficult to determine whether the addresses accessed by pointer operations will remain inside of an array, particularly since C performs no bounds checking on array operations. It is quite legal with C to write to the 11^{th} element of a 10 element array. While this is almost never intended, the reconfigurable application should implement the code as it was specified originally.

One possible solution to the pointer ambiguity problem is to simply disallow the use of pointers in hardware blocks. This might have the side effect of invalidating many blocks of code which would otherwise be ideal for hardware implementation. The programmer could be required to rewrite the pointer operations as array element accesses whenever possible, but this could require the programmer to modify large amounts of code. The best solution is ultimately to allow the coprocessor to access main memory. While cache coherency might be a problem, cache values could be invalidated just as they would be for DMA (direct memory accesses).

The final type of code the feasibility analyzer must watch for is the call to a non-hardware function. Assuming that the hardware model does not allow the coprocessor to call software functions on the host CPU, any block which contains a call to a function which either cannot be or is not implemented on hardware must be removed from consideration. There are four types of function calls:

- *I/O and Operating System functions.* These functions are inherently unsuitable for hardware implementation, and blocks containing these calls should be marked as unfeasible.
- *Functions for which source code is unavailable.* The functions are contained in linked object files, and are unavailable in source form. Since source code is unavailable, no hardware version can be developed. The block should be implemented in software.
- *Functions for which source is available, but are marked unfeasible.* The called function has been analyzed, and found unsuitable for hardware implementation. Since the function must be implemented with software code, the calling block must be implemented as software code as well.
- *Functions for which source is available, and are HW feasible.* In this case, both the block and the called function are hardware feasible. In this case, the calling block is feasible only if the called function is placed in hardware as well. This dependency reduces the partition space, since the partition which places the

calling block in hardware but the called function in software need not be examined.

Following feasibility analysis, all blocks which are marked as unfeasible are removed from consideration. Runtime and cost estimation is performed on the remaining blocks, which are then placed in either hardware or software by the partitioner. As shown earlier, each block marked as unfeasible for hardware implementation reduces the partition space by a factor of two. Effective feasibility analysis reduces the estimation and partitioning time considerably.

3.5 Estimation

The accuracy of the partitioning process depends on accurate estimates for the hardware and software characteristics of the blocks. Accurate estimates make it more likely that the partitioning algorithm will find the optimal partition, and that the resulting application will have the smallest possible execution time.

3.5.1 Software Runtime

Software runtime is one of the most important estimates that must be made for partitioning to succeed. Software runtime estimates are made for each block, and by combining estimates, for the entire program. For any individual block, comparison of the hardware and software runtimes will show whether that block would benefit from hardware implementation. Examination of the global runtimes with a block implemented in hardware versus software will show the block's effect on the overall runtime.

A variety of techniques can be used to estimate the software runtime of a code block or an entire application. To ensure that the estimates reflect the typical operation of the code as accurately as possible, *execution profiling* should be used to find the more frequently executed parts of the code. The code is instrumented, then executed with one or more sample data sets. Profiling probes record the number of times each block of code was executed. A much faster static examination of the code could be used to produce estimates of the total runtime, but it would not take into account the effects of actual input data which might cause one loop to be executed many times more than another. With execution profiling, a very accurate identification of computationally intensive portions of the code can be made, and the overall runtime estimated much more accurately. To reduce the possibility that the sample data set is not representative of the typical case, several profiling runs could be made and their results merged to form the basis for the runtime estimation.

The best method of software runtime estimation combines profiling information with the results of actual code generation. The reconfigurable compiler can generate the code for the software block quickly, and count the number of clock cycles required to execute it. Multiplying the estimate by the number of times the block was executed during the profiling run yields a total runtime for the block. Summing the estimates for all of the blocks in the application yields a global estimate.

There are several complicating factors with this type of estimation. It cannot take into account all of the effects of code optimization, since code could optimized inside of a

single block, but not between blocks. Runtime effects cannot be taken into account, such as the effects of cache misses and page faults on execution time. In addition, dynamically-scheduling superscalar processors make it more difficult to predict the execution of instructions. However, most of these difficulties could be overcome with sufficiently accurate estimators. Compilers for dynamically-scheduling superscalar processors often develop accurate estimates of execution times to facilitate code scheduling. The software runtime estimator would perform the same estimation, but instead of creating the final executable code, would simply count the number of clock cycles needed to execute it.

Other techniques can be used to estimate software runtime. The COSYMA system uses simulation instead of profiling. COSYMA constructs an Extended Syntax Graph representation of the code, which is simulated using input supplied by the user to obtain profiling information. This information is used to identify computationally intensive blocks of code, particularly those loops which comprise a large portion of execution [19:70]. The disadvantage with this system is that it is very slow, since the code is simulated by the compiler instead of executed at full speed. For this reason, the size of the simulations is limited in size, and thus the accuracy of the profiling information is limited.

3.5.2 Hardware Runtime

Hardware runtime estimation is perhaps the most critical estimate used in the partitioning process. The hardware runtime of a code block is defined as the time

required to execute the operations of the block with a hardware implementation, neglecting all of the overhead associated with configuring the hardware or transferring data back and forth between the coprocessor and the host. Since the partitioner will usually not consider blocks whose $t_{HWRT} > t_{SWRT}$, inaccurate estimation of the hardware runtime could remove blocks from the set of partitionable candidates which would benefit form hardware implementation. Hardware runtime is the core estimate around which the other estimates are placed.

As with the other types of estimates, the architectures used in modern FPGAs create unique difficulties for accurate timing estimation. These effects are discussed in Section 3.5.2.1. Limitations of high level programming languages create additional problems, since HLLs often cannot succinctly describe operations which have simple hardware representations. These problems are discussed in Section 3.5.2.2. However, certain opportunities do exist for the exploitation of parallelism in HLL code, which offer the hope for speedups over software implementations. Parallelism is discussed in Section 3.5.2.3.

3.5.2.1 FPGA Timing Estimation

The performance of FPGA designs is very difficult to estimate before the actual routing process is performed. Propagation delay in FPGAs consists primarily of three components: delay through the logic blocks, through the I/O blocks (including pad delay), and through the routing. Propagation delay through the logic blocks and I/O blocks is

typically well characterized, as shown by several typical delay values for the Xilinx XC4025-2 FPGA [53:62]:

Delay	Min Value	Max Value
CLB Lookup Table Delay (F/G Input to X/Y Outputs)		2.0 ns
CLB Sequential Logic Delay (Clock K to outputs Q)		2.8 ns
CLB Clock High Time	3.0 ns	
CLB Clock Low Time	3.0 ns	
IOB Output Delay (From Routing to Pad)		4.1 ns
IOB Input Delay (Through Pads to Connection to Routing)		3.0 ns

Table 4: Typical CLB and IOB Timing Parameters for Xilinx XC4025 FPGA [52]

Routing delays are the primary source of inaccuracy in FPGA hardware performance estimation. Signals connecting logic blocks to each other or to I/O blocks can pass through an arbitrary number of routing lines and an arbitrary number of switches. Different types of routing lines exist, each with different lengths and different capacitances. Any line segment could be connected to a number of different logic blocks, further increasing capacitance. In addition, the switch matrices are typically composed of pass transistors, which add additional capacitance to the circuit. All of these factors combine to great an often significant delay through the routing.

Most of these delay factors could be estimated, were it not for the additional difficulty of determining the exact path of any particular signal before routing is performed. For example, two logic blocks may be connected through the routing as shown on the left side of Figure 19. Another attempt to route the circuit may connect the blocks in a different path, as shown on the right side of the figure. The path on the right

passes through more switch matrices, and has a higher delay than the implementation on the left. This delay is impossible to estimate before routing is performed.

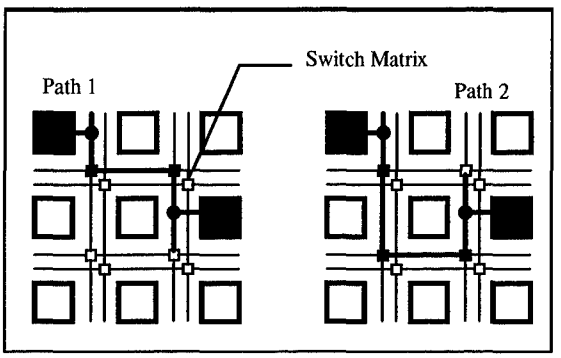

Figure 19:Two Possible Routing Paths Connecting Logic Blocks A and B

The hardware runtime of a block B is defined in the following equation. Propagation delay through the I/O blocks is ignored, since it is considered part of the communication time and not a factor in internal computation time.

$$t_{HWRT}(B) = t_{CLBs}(B) + t_{routjng}(B) \tag{9}$$

where $t_{CLBs}(B)$ is the sum of the delays along the longest path through the logic blocks making up B, and $t_{routing}(B)$ is the worst case delay through the routing.

Routing delay typically accounts for 40-60% of the total delay of an FPGA circuit [13:11]. Since a more accurate estimate cannot be determined without routing the circuit, a scaling factor will be used to derive routing delay as a percentage of the delay through the logic blocks.

$$t_{HWRT}(B) = t_{CLBs}(B) + t_{CLBs}(B) \times \alpha \quad (10)$$

In this expression, α is used as a scaling factor. If routing is assumed to be 50% of the total delay, $\alpha = 1$.

The hardware runtime estimate ultimately depends on the maximum number of CLBs through which the signals of code block B must pass. For a purely combinational block, equation (9) can be used unmodified. For sequential statements, such as *for* loops, the runtime of the statements in the loop body must be multiplied by the number of iterations of the loop. For sequential statements iterated *i(B)* times, the hardware runtime expression is:

$$t_{HWRT}(B) = \left(t_{CLBs}(B) + \alpha t_{CLBs}(B)\right) \times i(B) \quad (11)$$

Most FPGA development libraries provide a variety of macros for common structures, such as adders, counters, and shifters. In addition, some FPGAs provide additional high speed routing resources to connect adjoining logic blocks to accelerate carry propagation. In this case, performance estimates for blocks corresponding to common HLL operations such as the 32 bit integer add may be available. These estimates may be used in place of the previous logic block delay-based estimates. For a function composed entirely of hardware macros, delay can be computed as follows:

$$t_{HWRT}(B) = \sum_{m \in M} \left(t_{macro}(m) + \alpha t_{macro}(m) \right) \times i(B) \qquad (12)$$

where $t_{macro}(m)$ is the estimated propagation delay for macro m, and M is the set of all of the hardware macros used to construct B. This equation does not take into account the effects of logic optimization, as discussed in more detail in Section 3.6.2.2. Estimates based upon expression (11) should be more accurate.

3.5.2.2 Hardware Requirements Estimation

Hardware runtime estimates for a code block can essentially be constructed from estimates for the hardware implementation of each statement. Registers are instantiated to contain each of the variables used in a code block that must be stored (either because they are communicated to or from the CPU, or they are used across multiple iterations of a loop), and hardware modules are instantiated to perform the operations specified by the code. As an example, consider the code block in Figure 20.

```
int a,b,c;
....
a = 3 + c;
b = a + b - 17;
c = (d + e) << 2;
```

Figure 20: Source Code for Hardware Runtime Example

The code in this block is converted by the reconfigurable compiler into a parse tree representation, as shown in Figure 21.

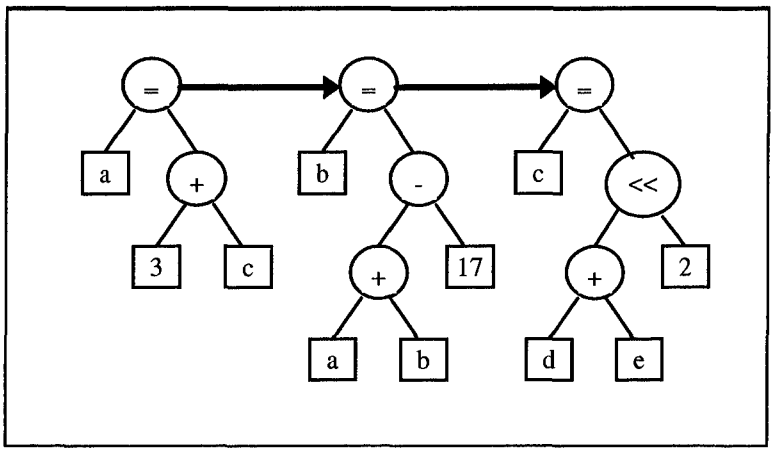

Figure 21: Parse Tree Representation of the Code Block

The first statement can be implemented with a single 32 bit adder. Statement 2 requires an adder and a subtractor, and statement 3 requires an adder and a shifter. Data dependencies can be identified and results passed directly to later statements as soon as they are available. The hardware representation of the circuit is shown in Figure 22.

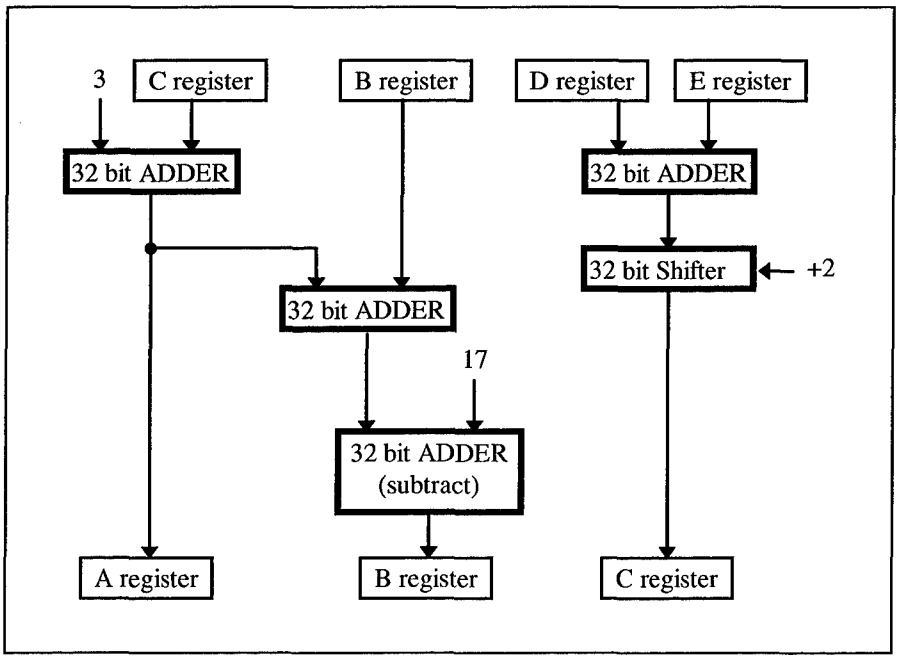

Figure 22: Hardware Implementation of the Code Block

The estimator would examine the hardware implementation and count the levels of logic blocks necessary to implement each statement of code. From this estimate, the estimate for the total hardware runtime for the block can be derived from the worst case path through the hardware. The estimate can be made before or after hardware optimization is performed The effects of hardware optimization are discussed in Section 3.6.2.2.

Hardware Module	Equivalent CLB levels
32 bit Adder/Subtractor	9
32 bit Shifter (shift left variable)	16
32 Bit Shifter (shift left constant)	0

Table 5: Equivalent CLB Levels for Operations Used In Example,

If a t_{CLB} = 2.0 ns and $\alpha = 1$, then the runtime for statement 1 is computed according to (9) as follows:

$$t_{HWRT}(B_1) = 9(2.0ns) + 9(2.0ns)1 = 36ns \tag{13}$$

Statement 2 places an adder and subtractor in series, and thus the runtime is twice that of statement 1:

$$t_{HWRT}(B_2) = 2\big(9(2.0ns) + 9(2.0ns)1\big) = 72ns \tag{14}$$

Since statement 3 shifts by a constant amount, the output lines of the adder used to compute the intermediate value (d+e) can be shifted two positions to connect to the

output, negating the need for shift hardware. Runtime is thus the same as for statement 1, or 36 ns.

A simple solution for the runtime of the entire block is to sum the runtimes of each statement, in this case 36+72+36 = 144ns. This does not take into account parallelism implicit in the statements, which allows several of the operations of the three statements to occur in parallel. When parallelism can be identified and exploited, runtime can be significantly reduced. Scheduling the operations in parallel whenever possible, the longest delay path represents the total runtime. As shown in Figure 22, the longest path is to compute B, through three 32 bit adders. The total delay is thus 3(9)(2.0) = 54ns. This type of estimate is more difficult to make, and requires much better dataflow analysis. Parallelism and its effects on runtime estimation are discussed in more detail in Section 3.5.2.3.

The method of runtime estimation needs to be expanded to account for sequential and conditional statements. Loop statements perform operations multiple times, while conditional statements can cause different operations to be performed based upon some conditional expression. Each type of operation is implemented in different ways, based upon methods for behavioral logic synthesis [15]. Each type of statement can add additional overhead to the runtime of the block.

Conditional statements such as *if-then-else* and *switch* constructs cause different operations to be performed in software program flow. The synthesized hardware will

perform the operations of all of the paths through the conditional statement in parallel, but only write the results of those operations selected by the conditional expression. An example statement is shown in Figure 23.

```
If (a==1)
   b = c + 2;
else
  b = b + 3;
```

Figure 23: Example Conditional Statement

The hardware implementation is shown in Figure 24. Both branches are computed, and the proper result for *b* is selected with a multiplexor, whose control is set based upon a comparator.

The hardware runtime of the *if* statement is thus the maximum runtime of the two branches plus an addition multiplexor and comparator delay for the conditional expression and statement overhead.

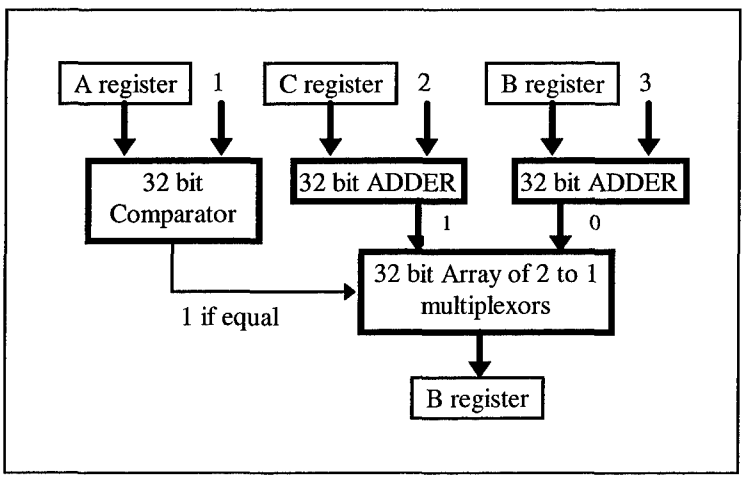

Figure 24: Hardware Implementation of the Conditional Statement

Loop statements require similar overhead. The structure of a loop statement is shown in Figure 25.

```
for (initialization expression; conditional expression; iteration expression)
  loop body
```

Figure 25: Structure of a For Loop

To implement a loop in hardware, a state machine is constructed. The conditional expression is used to generate a Done flag, which is detected by the hardware controller when the loop completes. The iteration expression is used to modify the loop control variable (usually to increment the value by some amount) between iterations. The initialization expression is used to set the loop control variables before execution commences. Finally, the loop body contains one or more statements which are executed during each iteration of the loop. In most cases, the body is composed of combinational logic, as shown earlier in this section.

Each variable in the loop is implemented as a register, whose value is written each clock cycle, dependent on the control signals generated by conditional statements in the loop body. *Break* and *Continue* statements disable the writes for all operations which occur after their location in the code. This can result in several possible results for each variable. Multiplexors select the proper results to write into the registers. In addition, *Break* statements cause the loop to exit early. This is implemented by combining the *break* signals with the Done flag generated by the conditional expression logic.

The unoptimized hardware implementation of the bit reversal loop is shown in Figure 26. The estimated timing effect for each statement can be added to compute a total estimate for the runtime of each iteration of the loop. Multiplication by the number of iterations of the loop yields a total hardware runtime.

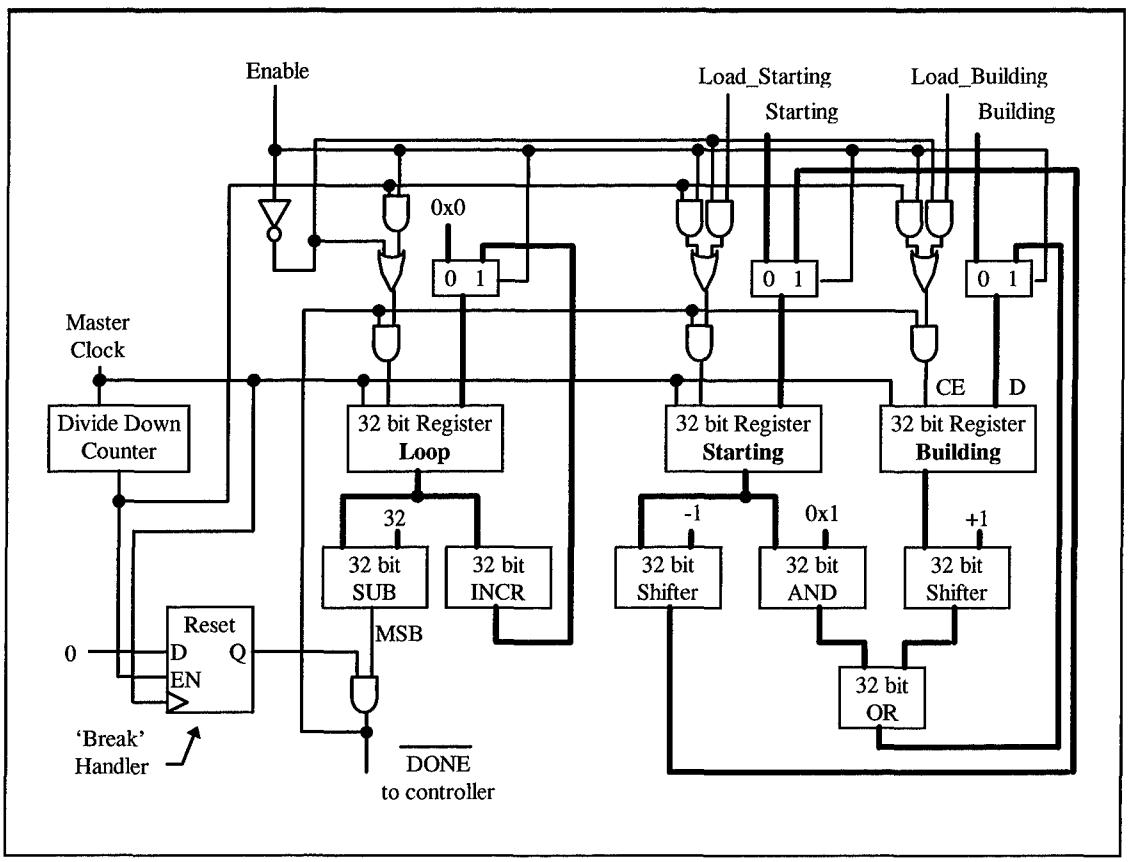

Figure 26: Hardware Implementation of the Bit Reversal Loop (Unoptimized)

3.5.2.3 Operation and Loop Level Parallelism

The hardware runtime estimation techniques described in the previous section ignore the possible benefits of parallelism. The runtime of each statement was estimated, and then summed to produce a final estimate for a block of code. Additional overhead was added for conditional and iterative statements. Operations and statements are

essentially executed one after the other, exactly as in a software implementation. Hardware implementation provides the possibility of executing operations and statements in parallel, significantly reducing the overall runtime.

The first opportunity for parallelism lies in scheduling the operations in a statement and in neighboring statements in parallel. For example, the code fragment in Figure 27, contains 3 statements. When scheduled with each operation in series, the first statement requires 3 operations, and the second and third statements require 1 operation each. The total runtime derives from 5 operations executing in series.

```
a = (b + 5) - (c + d);
e = b + 4;
f = a + 2;
```

Figure 27: Code Fragment With Opportunities for Parallel Operation Scheduling

The real opportunities for hardware speedup come when operations and statements are scheduled in parallel. Dataflow analysis is used to identify operations which can be performed at the same time. For example, the two additions of statement 1 can be performed in parallel, reducing the *delay height* of statement 1 to two operations. The addition operation of statement 2 can be performed at the same time, further reducing the height of the overall block. The addition operation of statement 3 depend on the result of statement one, however, so it cannot be scheduled in parallel. By scheduling the operations of the block in parallel, the overall delay height can be reduced to 3 operations, as shown in Figure 28. This method of operation scheduling is very similar to scheduling algorithms used in optimizing compilers, and the same dependence analysis can be used.

While similar to scheduling for superscalar processors, the reconfigurable coprocessor effectively can perform an unlimited number of operations in parallel, bounded only by the available hardware.

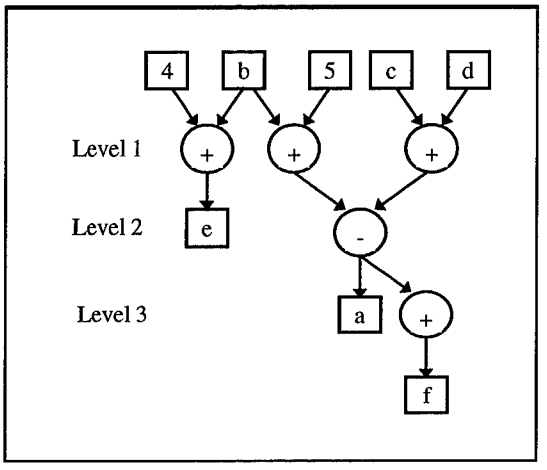

Figure 28: Operations of the Code Block When Parallel Scheduling is Used

The second opportunity for parallelism is between iterations of loops. Providing there are no cross iteration dependencies, successive iterations of a loop can all be performed in parallel. For loops which do have cross iteration dependencies, it still may be possible to pipeline iterations of the loops to achieve higher speedup using a hardware form of loop unrolling. The benefits of these techniques are limited only by the complexity of the scheduling algorithm used by the reconfigurable compiler.

A variety of research has been done to investigate hardware runtime estimation, primarily for ASIC-based codesign. Most efforts use path based estimation techniques similar to the ideas presented in this section. More information can be found in [27][54]. More information on scheduling and compiler optimization techniques can be found in [2].

3.5.3 Hardware Cost

Hardware cost is difficult to estimate accurately, due to the difficulties inherent in the architecture of FPGAs. As stated earlier, there are three components to hardware cost: logic blocks, I/O blocks, and routing resources. Logic blocks are used to implement the code block. I/O blocks are used to interface with the system bus, and with the other FPGAs of the coprocessor. Routing is used to connect logic blocks with each other and with I/O blocks. Since the amount of each resource is finite, accurate estimation of the hardware requirements for each code block are necessary to verify that multiple code blocks can be placed in the hardware at the same time.

The coprocessor consists of two components: a controller unit which interfaces with the system bus and handles communication with the rest of the system, and the programmable logic upon which hardware functions can be implemented. The controller contains logic interface with the system bus and to control the hardware functions on the FPGAs, passing them their operands and retrieving the results. The controller unit could be implemented from CLBs on one of the coprocessor's FPGAs, or could be implemented on a separate chip.

The resources required by the controller unit are static in nature, regardless of the hardware functions used. The resources used by the controller are permanently allocated, and are removed from the pool available to hardware functions. Therefore, only the logic used by the hardware function itself need be estimated. The hardware function includes

the logic necessary to implement the code block, plus any additional logic needed to interface with the controller.

The logic block requirements of a hardware block are the easiest to estimate, and can be in much the same way as hardware runtime. The statements in the block are analyzed to determine which operations are used, and a hardware module is instantiated for each. A lookup table is used to determine the logic block requirements for each operation. The logic block hardware cost for a simple code block *b* is thus:

$$c_{CLB}(b) = \sum_{o \in O} c_{CLB}(o) \tag{15}$$

where O is the set of all operations implemented in block *b*.

The hardware estimates can be adjusted to reflect logic optimization. For example, an unoptimized implementation of the statement *c = a & 0x1* would require the use of a 32 bit AND operation, composed of 16 CLBs (There are 2 lookup tables and outputs per Xilinx CLB). Optimization of the logic results in a circuit in which c[31:1] are tied to 0, and c[0] = a[0]. No logic blocks are needed at all.

I/O blocks are estimated in a different manner. There are two primary uses for IOBs: connection to the system bus and connection to other FPGAs or devices on the coprocessor. The IOBs used by the hardware controller to interface with the system bus are not counted for individual hardware functions. They are counted as overhead, and

removed from the pool of available resources before estimation begins. If the hardware function is on the same FPGA as the hardware controller, IOB cost can be ignored.

If the hardware controller is on a different FPGA, then IOBs must be used to connect the hardware function to the controller. If there are not enough IOBs available to pass all of the parameters and results in parallel, an additional controller must be implemented to serialize the data onto the available IOBs. In this situation, the additional logic required for this controller must be added to the cost of the hardware function. The expression for the IOB requirements of a hardware function is:

$$c_{IO}(b) = \sum_{v \in V} c_{IO}(v) \tag{16}$$

where V is the set of all variables passed to or retrieved from the hardware function. $c_{IO}(v)$ is the number of IOBs required for variable v.

A third, less common, use for the IOBs occurs when the hardware function requires several FPGAs to implement, as could be the case with larger functions. In this case, IOBs are needed to communicate between the parts of the hardware function itself. The exact number of IOBs required depends on the partitioning of the hardware function between FPGAs, and is extremely difficult to estimate without actually performing the entire partitioning, placement, and routing process. In this case, only rough estimates of IOB requirements can be made.

Routing estimation is extremely difficult. The routing is composed of many small metal interconnect lines and switch matrices. The exact requirements for a hardware function depend heavily on the routing algorithm used, and on the other logic in the design, including the hardware controller and other hardware functions destined for the same bitfile (See Section 3.5.5 for more information on the grouping of hardware functions into bitfiles). The paths used through the routing are unknown, as well as the placement of the logic blocks on the FPGA. The unknowns in this situation are so great that no method has been found to estimate the routing requirements of a hardware function without performing the actual process.

Rather than estimate the routing requirements, a small modification on the methods used to estimate the logic block requirements will increase the probability that the design can be routed. A scaling factor could be adopted to cause the logic block requirements for a hardware function to be overestimated. This is an adaptation of the FPGA rule of thumb "Use no more than 70% of the stated gate capacity of the FPGA" to ensure the design will route. The new equation for logic block cost for block b is:

$$c_{CLB}(b) = \omega \times \sum_{o \in O} c_{CLB}(o) \tag{17}$$

where ω is some scaling factor > 1.

All of the estimates can be made more accurate following an actual routing cycle. Information obtained through back annotation can be used to find the actual values for

each cost measure. Incremental routing algorithms, if used, will minimize the change in hardware cost (and performance) between iterations, so long as the partitions are not too widely different. Back annotation is discussed in more detail in Section 3.7.

Other methods of hardware cost estimation have been proposed. In [45], an incremental approach is used to analyze the number of data and controls signals, datapath units, and other hardware characteristics to quickly update estimates for a hardware/software codesign system. Targeted towards ASICs, the authors report an error in estimation of only 7%. Tests of the technique on FPGAs yield a larger variation, but the results are promising. Most present codesign research utilize much simpler methods of estimation. Hardware cost in the COSYMA system is estimated implicitly by the number of basic blocks moved to hardware [28:2]. Since basic blocks can require widely varying amounts of logic to implement, the hardware estimates used in COSYMA cannot be used to reliably estimate if a design can be placed upon the programmable logic devices used by the hardware.

While the methods to estimate hardware cost are imprecise, hardware cost is a secondary consideration to the partitioning process. Hardware cost is primarily used as a means of estimating whether a particular partition will fit in the available hardware before synthesis is performed. Inaccurate hardware cost estimates can be updated during the back annotation stage, and later estimates are likely to be more accurate. The scaling factor, ω, provides a means to alter the aggressiveness of the partitioning process in packing hardware functions onto the coprocessor.

3.5.4 Communication Time

Communication time is defined as the time required to pass the operands of the hardware function to the coprocessor, and to retrieve the results following execution. Communication time for a block B can thus be written as follows:

$$t_{comm}(B) = t_{comm\_operands}(B) + t_{comm\_results}(B) \tag{18}$$

Define Comm_To(B) as the set of variables which are operands to block B and must be sent to the coprocessor, and Comm_From(B) as the set of variables which are results of block B and must be retrieved from the coprocessor. Define N as the bandwidth of the bus in bytes per second. If sizeof(Comm_To(B)) returns the number of bytes comprising the variables in Comm_To(B), t_{comm} can be written as follows:

$$t_{comm}(B) = \frac{\left(sizeof(Comm\_To(B)) + sizeof(Comm\_From(B))\right)}{N} \tag{19}$$

The sets Comm_To and Comm_From are computed using dataflow analysis of the code. An introduction to dataflow analysis can be found in [2:608-680]. Aho develops the concept of Live In and Live Out sets. The Live In set for a block B is the set of all variables which have valid definitions at the point in the code immediately preceding entry into block B. The Live Out set for block B is the set of all variables which have valid definitions at the point in the code immediately following exit from block B, and are referenced at some point later in the code.

```
a = 5;
b = a + 3;
c[0] = 4;
d = a + b;
e = a;
for (loop = 0; loop < 10; loop++) {
  c[loop] = loop +  b;
  x = loop + c[loop];
}
printf("%d %d %d % d", c[5],a,b, e)
```

Figure 29: Code for Live In and Live Out Set Example

In the following example, the partitioner is estimating the communication time for the *for* loop, called block B. The Live In set for B, denoted *in*(B), is {a,b,c,e}, since those variables have valid definitions at the start of block B and are used later in the code. Note that variables *x* and *d* are not live in, since no definition exists before block B for *x*, and there are no further uses of *d*. The Live Out set for B, denoted *out*(B), is {a,b,c,e}, since only those four variables are referred to later in the code.

The estimation program needs to determine which variables must be sent to block B, and which variables must be retrieved from it. One solution is to send the entire Live In set to the hardware, and retrieve the entire Live Out set. This would be inefficient, since the system would send the hardware variables it does not reference, and retrieve variables which are never used again. For example, *a* is part of *in*(B), but it is not used by the loop. Variable *e* is both live in and live out of the loop, but is not used. It is simply passed through to the next block of code after the loop.

Instead of using Live In and Live Out in isolation, two other sets are developed to reflect the actual uses of the variables inside block B. Let Cum_Def(B) be the set of all

variables which are defined inside block B, called the *cumulative definition* set for B. Let Cum_Use(B) be the set of variables which are referenced inside block B, called the *cumulative use* set for B. Cum_Def(B) for the example is {c,x, loop}. Cum_Use(B) is {loop, b, c}.

Comm_To(B) is defined formally as the intersection of the live in and cumulative use sets:

$$Comm\_To(B) = In(B) \cap Cum\_Use(B) \tag{20}$$

Comm_From(B) is defined as the intersection of the live out and cumulative definition sets:

$$Comm\_From(B) = Out(B) \cap Cum\_Def(B) \tag{21}$$

For the example in Figure 29, Comm_To(B) = {a,b,c,e}∩{loop, b, c} = {b,c}. Comm_From(B) = {a,b,c,e}∩{c,x, loop}= {c}.

Note that the array *c* is retrieved from the hardware in its entirety, even though c[5] is the only element accessed in later code. If the dataflow analysis used by the reconfigurable compiler can track individual elements in the array, then only the affected elements need be passed, further reducing communication time. Since the use of pointers and other means of accessing arrays can make it difficult to determine at compile time which elements will be modified, it is safer to pass the entire array to and from hardware.

For most arrays, most of the elements will be modified, and the extra elements will represent only a small fraction of the overall communication time.

Another problem that makes partitioning more difficult is the dependence of the estimates for one block on the placement of other blocks. In simpler terms, the estimates for a block can change depending on how other blocks are partitioned, and cannot be computed in isolation. This is most noticeable in communication time. For example, consider three adjoining code blocks A,B, and C as shown in Figure 30.

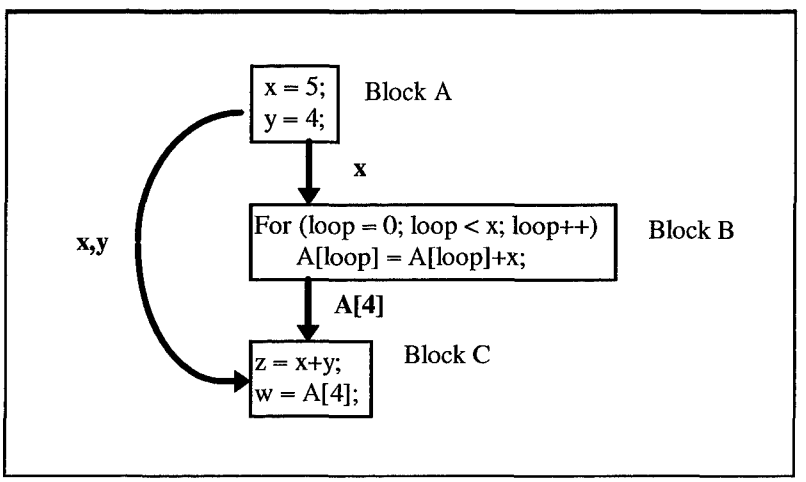

Figure 30: Dependence of t_{comm} on Adjoining code Blocks

The black lines represent data which must be passed between the blocks. If block B is placed in hardware, then variable *x* must be passed to the hardware, and variable *A[4]* must be returned to block C. This is shown in Figure 31.

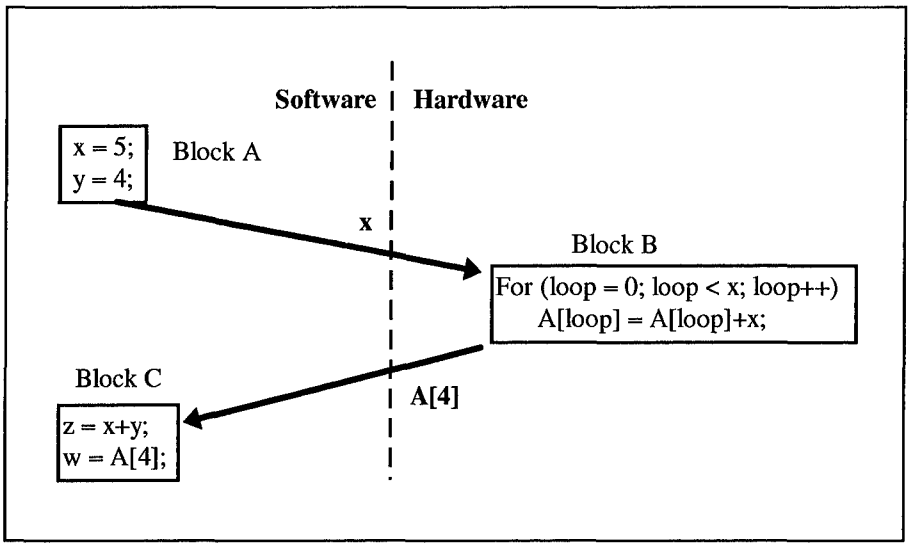

Figure 31: One Possible Partitioning of the Blocks, Showing Communication Requirements

If the partitioning is changed so that blocks A and B are both implemented as hardware functions, then only variable A[4] needs to be passed. Block B no longer needs to pass variable *x*, and the communication time estimate is thus reduced. This new partitioning is shown in Figure 32.

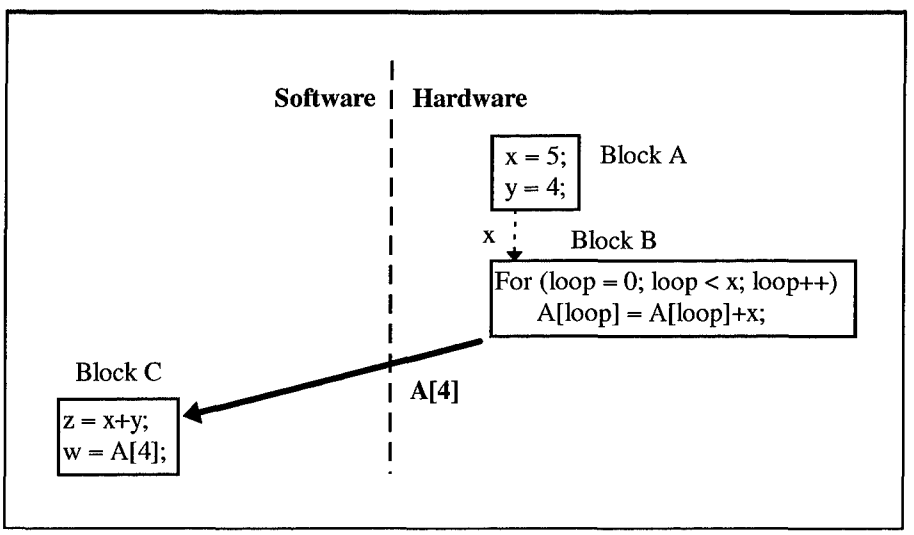

Figure 32: A Second Partition of the Code, Reducing the Communication Requirements

The overall effect of the problem is to cause the technique discussed in this section to overestimate the communication requirements for blocks which use variables also used by other hardware blocks. Since communication time is often the largest fraction of overall hardware execution time, this can severely impact proper partitioning. There are two solutions to the problem. The first is to require the partitioner to recompute communication time for every block with each partition so that the communication times reflect the mapping of blocks to hardware, while the second is to adopt a faster compromise solution. Two communication times are stored for each block. The first is the ceiling estimate as computed earlier, and the second assumes that the adjoining blocks are also mapped to hardware. The other uses of the variables in B are most likely near the block, due to locality of reference, and thus the communication time will likely be reduced to a level near what it would be if the estimates were recomputed for each partition.

The former solution is the most accurate, but requires the estimates to be recomputed for each test partition, possibly slowing the partitioning process significantly. The second solution is presented as a faster, compromise solution. Given the increasing difference in speed between CPU and bus clock speeds, communication will represent an ever increasing percentage of the total hardware execution time. It is possible that first solution is worth the extra computation time because it will not tend to overestimate the communication time more than necessary, and thus produce better partitions.

3.5.5 Configuration Time

Configuration time for a hardware block B is defined as the time required to program the FPGAs of the coprocessor to implement that block. It typically includes the time to program the look-up tables inside the CLBs and program the registers which control the pass transistors connecting the routing resources throughout the FPGA. Additional checksum and addressing bits may also be included in the bitfiles as well. Specific information on FPGA configuration can be found in [36][6][52]. Configuration of the FPGAs in the coprocessor depends on the architecture of the FPGAs. Two common types of reconfiguration exist: total configuration, and partial configuration.

3.5.5.1 Total Configuration

As discussed in Chapter 2, most current FPGA architectures do not allow partial reconfiguration of the device. The entire FPGA must be configured at once. This has several ramifications for a reconfigurable system. FPGAs typically take several milliseconds to configured completely, as shown by the examples in Table 6. For a processor running with an internal clock speed of 100MHz, the largest FPGA shown requires over 14 million clock cycles to configure.

FPGA	Gates	Bits	Config Time @ 10 MHz
XC4002A	2,000	31,628	3.17 ms
XC4010/D	10,000	178,096	17.8 ms
XC4025	25,000	422,128	42.2 ms
XC4028EX	28,000	668,127	44.8 ms
XC4036EX	36,000	832,483	83.2 ms
XC4044EX	44,000	1,014,879	101 ms
XC4052XL	52,000	1,215,323	122 ms
XC4062XL	62,000	1,433,847	143 ms

Table 6: Configuration Times for Typical FPGAs

With configuration times as large as these, reconfiguration should be performed as rarely as possible. Whenever possible, multiple hardware functions should be configured at the same time, so that the configuration overhead is amortized over as many functions as possible. If all of the hardware functions can be contained in a single bitfile, this poses no problem. But if multiple bitfiles must be used, deciding which functions should be placed in each bitfile can be a problem. Bitfiles are created for combinations of hardware functions. The partitioner already schedules hardware functions, so it is usually possible to determine which hardware functions are likely to be on the hardware at any given instant.

One problem with multiple function bitfiles is that program execution is not predictable, and situations may occur in which a hardware function is called unexpectedly. For example, consider an application which has 9 hardware functions, named A, B, C... I. The expected execution order of the functions based upon profiling is ABCBDEFGHI. Seeking to minimize the number of configurations, the reconfigurable compiler creates three bitfiles, containing functions ABC, DEF, and GHI. If execution

occurs as expected, three reconfigurations are needed, as shown in the top half of Figure 33.

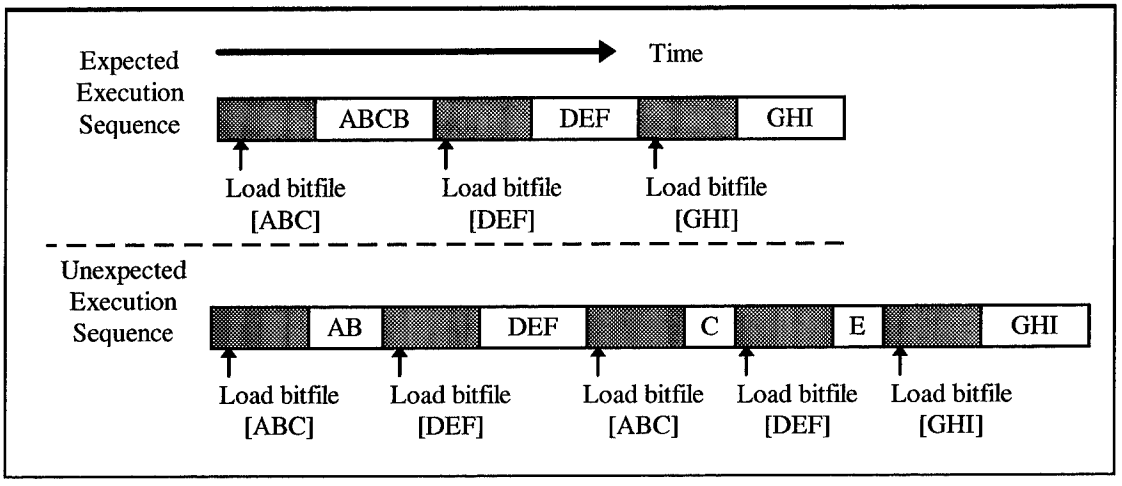

Figure 33: Inefficient Grouping of Functions into Bitfiles, Resulting in Unnecessary Reconfiguration

If the function execution order is different from the expected, additional loads may be required. If the actual execution order is ABDEFCEGHI, five reconfigurations of the hardware are required, as shown in the bottom half of Figure 33. Additional reconfigurations can significantly increase overall execution time.

A solution is to create multiple bitfiles, containing combinations of hardware functions. For the previous example, two additional bitfiles could be created containing functions ABD and CEF. With these additional bitfiles available, only three reconfigurations would be required in both cases, as shown in Figure 34.

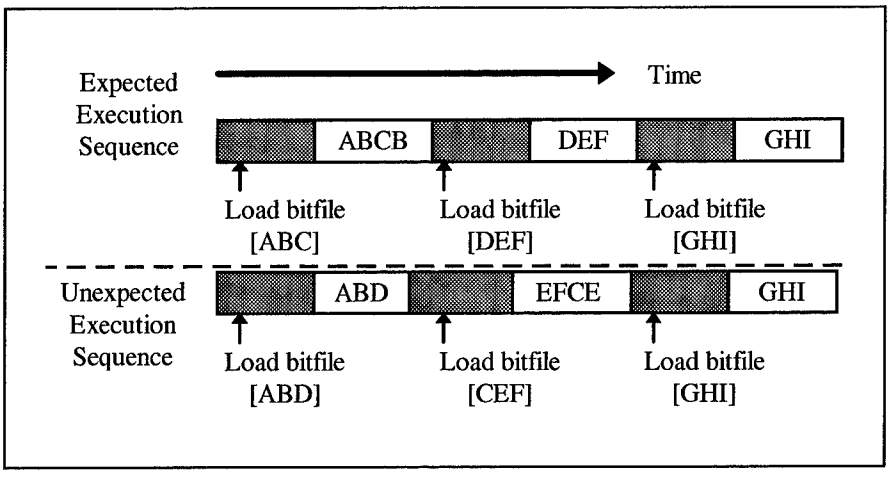

Figure 34: Reduced Reconfiguration Needed Due to Redundant Bitfiles

As many additional bitfiles can be created as are desired, limited by the amount of disk space available for use. While functions can be grouped together arbitrarily, execution paths in the code can be analyzed to determine intelligent groupings of hardware functions. For example, two hardware functions in the same *if-then* path are likely to be executed together, and can be grouped accordingly. The optimal number of bitfiles is dependent on the exact nature of the code, and may benefit from future research.

The instantaneous configuration time for a hardware block *b*, for a specific call of that block, is defined as shown:

$$t_{config}(b) = \left(\frac{x}{S} + x \times \theta_{os}\right) \times \frac{config\_bits(b)}{\sum_{f \in C} i(f)\, config\_bits(f)} \qquad (22)$$

where *S* is the configuration speed in bits/second, *x* is the number of bits in a bitfile, config_bits(b) is the number of configurations bits for hardware block b, and *i(f)* is

the number of times block *f* is invoked when this bitfile is loaded. θ_{os} is the operating system overhead to read the bitfile and prepare the FPGAs for configuration.

Using the previous example, the configuration time for block E is computed as follows. Assuming that the original three bitfiles are used, ABC, DEF, and GHI, five reconfigurations of the hardware are needed. With the execution order AB**DEFCE**GHI, function E is called twice. Each call has a different configuration time. During the first load of the bitfile, all three functions are called, as shown in the following table. The FPGA requires 200,000 configuration bits, divided among the three functions as shown.

Function	Config Bits	Times Called
D	40,000	1
E	100,000	1
F	60,000	1

Table 7: Configuration Information for the First Load of Bitfile Containing E

Given the configuration speed of the FPGA is 10Mbits/sec (a typical speed for Xilinx and Atmel FPGAs), the configuration time for the first call to E, $t_{config}(E_1)$, is computed as follows. For simplicity, θ_{os} = 12.5ns/bit (10 MB/sec disk access).

$$t_{config}(E_1) = \left(\frac{200Kbits}{10Mbits/\sec} + 200Kbits \times 12.5ns/bit \right) \times \frac{100Kbits}{(40+100+60)Kbits} = 11.25m\sec \quad (23)$$

For the second call to function E, the other functions D and F are not called. The configuration information is shown in Table 8.

Function	Config Bits	Times Called
D	40,000	0
E	100,000	1
F	60,000	0

Table 8: Configuration Information for the Second Call to Function E

The entire penalty for loading the bitfile is placed upon function E. $t_{config}(E_2)$ is computed as follows:

$$t_{config}(E_2) = \left(\frac{200Kbits}{10Mbits/\sec} + 200Kbits \times 12.5ns/bit \right) \times \frac{100Kbits}{100Kbits} = 22.5m\sec \qquad (24)$$

The configuration time for the second call to function E is significantly higher than for the first. It is possible that runtime may be lower if a software implementation of block E is used for the second call and a hardware implementation is used for this first. Multiple versions of code blocks adds an additional level of complexity to the partitioning process. To make the best partitioning decision for a specific call to a code block, the bitfile contents must be known, since the configuration time estimates will change, depending on the contents of each bitfile.

A slightly simpler alternative would be to create a single estimate for the configuration time of a block, based upon an average of all of the configuration times for that block over the entire execution of the program. A simple grouping of functions into bitfiles would be performed (before actual logic synthesis is performed, so the results

would be speculative). Using the previous example, the functions were grouped into bitfiles as ABC, DEF, and GHI based upon estimates of the resources used for each function and their probable execution order. At this point, a single estimate for the configuration time for E can be derived as the average of the two calls to the function during the profiling run. Averaging the two configuration times together yields a single estimate for $t_{config}(E)$= (22.5ms +11.25ms)/2 = 16.9ms. This average value would be used instead. The equation for the average configuration time for a block is shown below:

$$t_{config}(B) = \frac{\sum_{f \in F} t_{config}(B_f)}{\#F} \tag{25}$$

where F is the set of all calls to hardware function B, # F is the number of elements in F, and $t_{config}(B_f)$ is the configuration time for the *f*th call to function B, derived from Equation (22).

3.5.5.2 Partial Configuration

The estimation of configuration time is much less complicated for FPGAs which allow partial reconfiguration. While most current FPGA architectures do not allow partial reconfiguration, newer architectures are incorporating this feature as manufacturers begin to produce devices specifically for reconfigurable computing applications (e.g. the Xilinx XC6200 series FPGA). Since each hardware function can be loaded individually, many of the problems discussed in the previous section do not apply.

Most FPGAs are configured in bit serial fashion. The configuration registers in an FPGA are connected in serial, in a manner very similar to the registers in modern boundary scan testing. The serial connection of logic blocks can be see in Figure 35. Even the FPGAs which allow byte parallel configuration serialize the bits before they are fed into the configuration registers.

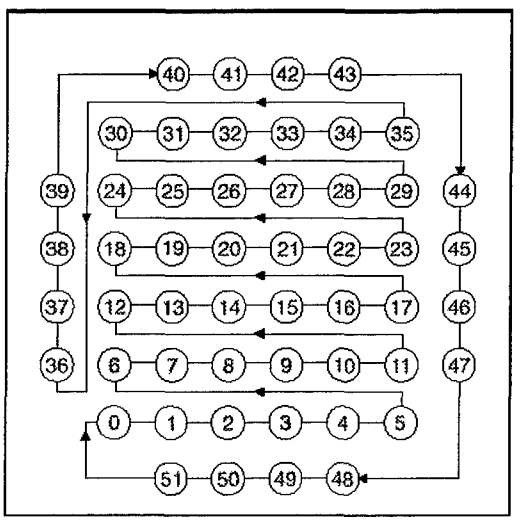

Figure 35: Serial Connection of Configuration Registers in an Atmel FPGA

A typical bitfile contains a header, a data section, and a trailer. The header contains several startup bits and a field indicating how many data bits are to follow. The data section contains the bits to configure the lookup tables in each logic block and the pass transistors in each switch matrix. The trailer typically contains checksum information to verify correct configuration of the device. Devices allowing partial reconfiguration typically modify the bitfile format slightly to break the data section up into one or more blocks, each of which has a starting address and an ending address, followed by the data to configure the logic blocks and routing switches in that address range [6:2-27]. An illustration of a typical bitfile for an Atmel FPGA is shown in Figure 36.

```
10101010  ;Preamble
00000000
00000100 ; Number of data segments
00000000
00000100 ; Start address of Segment 1
00000000
00001100 ; End address of Segment 1
11010111  ; Data
11101011
 ........
00000000
00010100 ; Start address of Segment 2
00000000
00100100 ; End address of Segment 2
01011011  ; Data
```

Figure 36: Typical Bitfile for a Partially Configurable FPGA

The number of configuration bits needed for hardware block B is defined as

$$b(B) = \rho + \sum_{c \in C} \left(a + \left(clbs(c) \times b_{CLB} \right) + \left(iobs(c) \times b_{IOB} \right) + \left(rout(c) \times b_{routng} \right) \right) \quad (26)$$

where ρ is the number of overhead bits in the header, *a* is the number of bits in the starting and ending address fields for each block. C is a set containing the sets of contiguous groups of logic blocks. Each member of the set *c* is a set of contiguous logic blocks which form one block in the bitfile (e.g. {9,10,11}, {15,16,17}, {20,21,22,23}, and {45,46,47} are the four members of C for the previous example). *clbs(c)* is the number of logic blocks in *c*, *iobs(c)* is the number of I/O blocks, and *rout*(*c*) is the number of switch matrices. The number of bits required to configure a single member of each type of resource are b_{CLB}, b_{IOB}, and $b_{routing}$.

Once the number of bits in the configuration file are known, the expression for the configuration time for block B is derived as follows:

$$t_{config}(B) = \frac{b(B)}{S} + \left(b(B) \times \theta_{OS}\right) \tag{27}$$

where S is the configuration speed of the device in bits/sec, and θ_{os} is the operating system overhead in seconds/bit.

The previous expression assumes that the hardware function B must be configured each time it is called. As shown in the previous section, it is possible for the same function to be called several times between configurations. Part of the task of the operating system is to determine if the hardware function is present on the coprocessor, and to only configure the hardware if necessary. In this case, the configuration time can be spread out among several calls to that function. If the partitioner performs scheduling

to determine the expected number of calls for block B before it is reconfigured, t_{config} for a particular block can be divided by the number of times that block is executed before it is removed from the hardware. The resulting equation for t_{config} is shown below:

$$t_{config}(B) = \left(\frac{b(B)}{S} + \left(b(B) \times \theta_{OS} \right) \right) \times \frac{1}{i(B)} \tag{28}$$

where $i(B)$ is the number of times block B is executed.

3.6 Synthesis

After the partitioner has identified those blocks which are to be implemented as hardware functions, a second pass is made over the source code to perform the actual extraction. The single source is broken at this point into two different specifications: the software specification, which will be compiled normally, and the hardware specification, for which logic will be synthesized, and the routing process performed.

3.6.1 Software Code Generation

The best way to interface the software code to the newly created hardware functions is through operating system calls. The operating system on the reconfigurable system would handle the specifics of FPGA configuration and communication, providing a consistent interface to the reconfigurable compiler and the user. The code for a hardware block would be removed, to be replaced by a call to the operating system passing both the new function name and the parameters of the block (identified during

communication analysis) as parameters to the function. An example of the bit reversal loop used earlier was included as Figure 9.

Interface generation for the software code is thus relatively straightforward. The Comm_To set of variables are passed by value, while the Comm_From set of variables are passed by reference. The results are placed in the proper memory locations by the operating system after it reads them from the coprocessor. Once all of the blocks marked for hardware implementation are replaced, the program can be compiled as it would be on a normal system. The operating system functions would come from a library, to be linked in later.

The tasks of the operating system functions are straightforward. The first parameter to the operating system function is the name of the hardware block to be executed. The operating system maintains a list of those functions for which the FPGAs of the coprocessor are currently configured. If the desired function is present on the hardware, configuration does not have to be performed, and execution can commence as described in Section 3.2.3. If the coprocessor is not configured to execute the desired function, the operating system loads the bitfile for that function into memory and configures the FPGAs over the system bus. From that point, execution continues normally.

3.6.2 Hardware Generation

3.6.2.1 Overview

Generation of the hardware for extracted blocks is slightly more complicated than for those blocks placed in software. A digital logic implementation of the behavioral HLL code for the code block must be generated. The reconfigurable compiler can generate the logic directly from its internal representation of the hardware block. Alternately, it can output the block as a behavioral hardware description language representation, and use a commercial logic synthesis tool to create the digital logic from which the FPGA bitfiles can be generated.

The hardware controller must be inserted into the design as well. The controller's task, as described earlier, is to interface with the system bus and control the operation of the different hardware functions which may inhabit the FPGA at any given time. The logic for the controller may be pre-routed, requiring only the connections to the hardware functions to be routed. Alternately, the controller may be specified in a HDL, to be synthesized with the hardware functions.

In systems which allow partial reconfiguration of the FPGAs, the hardware controller can be configured on the FPGAs upon startup. Only the hardware functions need be reconfigured, leaving the controller to operate continuously throughout program execution. In systems which do not allow partial reconfiguration, entire FPGAs will have to be reconfigured at once. If the controller is on the same FPGA as a new hardware function, it will be written over with a new controller. This overhead is added to the

configuration time for the hardware functions. To amortize the cost of configuration, however, as many hardware functions as possible will be configured at one time, to spread out the configuration overhead as much as possible. These groups will have been identified by the partitioner. Instead of having configuration bitfiles for individual hardware functions, bitfiles are created for groups of hardware functions.

In any case, creation of logic from behavioral descriptions is relatively straightforward. Commercial packages such as Synopsys [15] and Cadence support synthesis of logic from behavioral HDL descriptions. Most of the code in a HLL description can be translated directly into an HDL description and synthesized. The hardware feasibility analysis identified blocks for which hardware generation would be difficult. The problem is simply one of translation between C and the HDL.

3.6.2.2 Hardware Optimization

A significant problem involved in the synthesis process is the inadequacy of C or other HLLs to represent simple hardware operations succinctly. As discussed in Section 3.5.2, HLLs often must represent simple hardware operations such as the bit reversal in terms of loops or other complicated language structures. The problem of identifying code which could be replaced with simple hardware structures is difficult. Other problems occur in terms of optimization of the widths of datapaths and hardware operations. If a loop is intended to perform 10 iterations (which can be represented with 4 bits), a 32 bit counter and datapath is a significant waste of hardware area. The resulting circuit also

operates at a lower speed than might otherwise be possible. Detection of these chances for optimization can be difficult.

One approach is to simply implement the hardware as specified. No effort if made to identify variables which can be represented with fewer bits or to identify the intent behind a block of code. All variables and operations with the same data type are the same width, regardless of bounds on their use. Loops like the bit reversal loop are implemented as loops, ignoring the much simpler implementation in which the input wires are connected to the output wires in reverse order. Hardware generated in this manner will produce correct results, but at a significant penalty in possible speedup. Much of the benefit of hardware implementation can be lost.

A different solution is to implement some form of code recognition. The reconfigurable compiler would examine the code intended for hardware implementation for variables or operations upon which optimization could be performed. There are two types of optimization: *operation and datapath reduction*, and *algorithm identification and replacement.*

Operation and datapath reduction searches for variables which contain values in a range smaller than that of their data types. For example, the variable *loop* in the bit reversal loop shown in Figure 7 really only uses the range 0 through 31. This range can be represented with 5 bits, a significant reduction in the 32 bits allocated to integers in many systems. If the reconfigurable compiler can identify this instance, it can reduce the

number of flip-flops needed to store the variable, replace the 32 bit adder with a 5 bit adder, and reduce the datapath in this part of the circuit from 32 to 5 wires. These changes can have significant effects on hardware cost and performance. This optimization can only be performed on variables for which the reconfigurable compiler is certain about the ranges of values, and is not appropriate for all variables.

Algorithm identification and replacement requires the reconfigurable compiler to identify what the programmer intended to accomplish with a certain block of code and replace it with a simpler hardware circuit that produces the same result. Unfortunately, this is an extremely difficult problem, for which no solution is known. Other alternatives could be used to achieve some of the benefits of efficient hardware circuits. A function library could be created, containing a collection of related functions in areas such as signal processing, encryption, etc. A function such as bit reversal would have two implementations, one in hardware and one in software. The partitioner would pick the appropriate version at compile time.

3.7 Back Annotation

3.7.1 Estimate Verification

Back annotation is an optional step in the compilation process, intended to produce more efficient results. Back annotation is performed once partitioning, placement, and routing is completed. Back annotation uses the results of the routing process to update and verify the estimates used in the partitioning process. While the performance of FPGAs can be very difficult to estimate before routing is performed, it is

possible to take the routed circuit and *back annotate* the actual timing values onto the different parts of the circuit. The path of each signal through the routing, including the number of pass transistor routing matrices the signal goes through is known, and delay can be calculated quite accurately. This information can be used to update the timing and hardware cost estimates for all of the blocks placed in hardware by the partitioner.

Ideally, the estimates for hardware runtime and cost made earlier in the process were accurate. If this was the case, then the partition produced by the partitioner was close to the best possible partition, and the application should perform very well. If the estimates were inaccurate, then the results of the partitioning were based upon invalid information, and the partitioned application may not be the best one possible. In this case, the estimates can be updated and the partitioning process reiterated to produce a better partition.

Parameters are specified for the reconfigurable compiler for maximum percentage error in hardware runtime, ε_{HWRT}, hardware cost, ε_{CLB}, ε_{IOB}, and ε_{R}, and configuration time, ε_{config}. Estimate checking is not performed on software runtime or on communication time, since they are not affected by variations in FPGA timing. For each block, the actual hardware runtime, ϕ_{HWRT}, actual configuration time, ϕ_{config}, and hardware costs, φ_{CLB}, φ_{IOB}, and φ_{R}, are computed based upon the routing results.

The error between the estimates and the actual values are computed for each block, b, and compared against the maximum error parameters as follows:

$$\frac{\phi_{HWRT}(b) - t_{HWRT}(b)}{\phi_{HWRT}(b)} \leq \varepsilon_{HWRT} \tag{29}$$

$$\frac{\phi_{config}(b) - t_{config}(b)}{\phi_{config}(b)} \leq \varepsilon_{config} \tag{30}$$

$$\frac{\varphi_{CLB}(b) - c_{CLB}(b)}{\varphi_{CLB}(b)} \leq \varepsilon_{CLB} \tag{31}$$

$$\frac{\varphi_{IOB}(b) - c_{IOB}(b)}{\varphi_{IOB}(b)} \leq \varepsilon_{IOB} \tag{32}$$

$$\frac{\varphi_{R}(b) - c_{R}(b)}{\varphi_{R}(b)} \leq \varepsilon_{R} \tag{33}$$

If the errors for the estimates are sufficiently small, then the partition is accepted and the process is complete. If the error in one or more of the estimates is too great, then the estimate must be updated. There are two ways to perform the estimate update. The estimate value can simply be changed to the actual value, or the methods used to make estimates for that type of code can be updated globally. For example, if the affected block is a *for* loop, loop overhead affects all *for* loops, not just the current one. The amount of logic needed for overhead, and the effects on timing can be used to update the estimates for all future *for* loops. The learning effect can be applied to all types of code.

An additional problem which arises during later iterations of the partitioning process is that the estimates are likely to change again due to the differences in routing. For example, if block A is placed in hardware during two successive partitioning runs, differences in routing of other blocks can change the actual timing results, even if the block is kept in the same place on the FPGA during routing. This can occur due to changes in the routing paths required when other signals are routed through paths previously used by block A's routing. This effect is shown Figure 37.

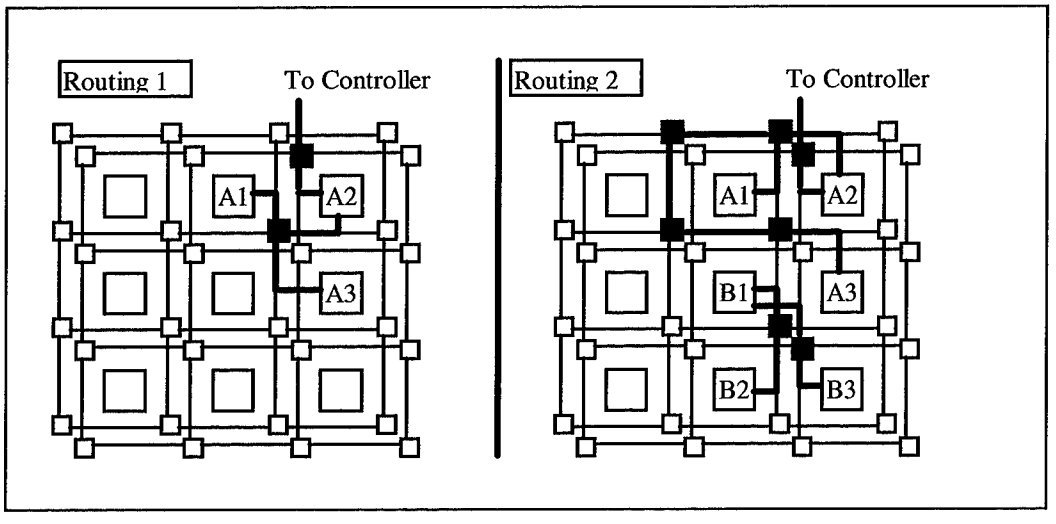

Figure 37: Changes to Timing of Function A Due to Routing of Function B

The most profound effect of this problem is that estimates may be updated every iteration through the outer loop of the development process, and the error may never stabilize. This might result in an infinite loop in the partitioning process. Once again, several solutions exist. The least complex is to simply limit the maximum number of times the partitioning process is performed. The second uses the concept of *incremental routing*.

3.7.2 Incremental Routing

To limit the variation in performance and hardware cost between different implementation of the hardware block, the interference of other blocks must be reduced. This can be done by require the routing system to avoid whenever possible changing the placement and routing of certain blocks when routing new ones. By routing new hardware blocks around existing ones, the timing of the existing blocks remains unchanged, and the time required to route the design is reduced.

To perform incremental routing, the router must be aware of which logic blocks and routing resources correspond to hardware functions. The router would try to keep the logic blocks and routing for a hardware function as contiguous as possible, and to avoid routing the signals for other functions through routing between the logic blocks of that function. Alternately, the router would be able to route other signals between the logic blocks of a hardware function, but only after all of the routing for that function has been completed. The overall effect would be to reduce the variation in paths taken through the routing for a hardware function, and thus the variation in performance between iterations of the outer partitioning loop.

It is not always possible to place the logic blocks of a hardware function contiguously and to avoid routing signals through other functions, particularly for those signals connecting to the hardware controller or to IOBs to connect to other FPGAs. In this situation, some amount of variance in performance is inevitable. The maximum

number of iterations of the partitioning process should still be used to ensure an infinite loop does not result.

The disadvantage of using incremental routing is that special routing algorithms must be used. Standard commercial routing software may be inappropriate, requiring special routing software to be created as part of the reconfigurable compiler. Since routing is closely related to the specific FPGA architecture used, portability of the reconfigurable compiler can be reduced.

3.8 Conclusion

This chapter created a framework for a reconfigurable compiler. The stages in the development process were identified, and the differences from a conventional compiler discussed. While similar in some ways to a conventional compiler, the partitioning and synthesis tasks are unique to reconfigurable systems development. The partitioning and synthesis problems were discussed in some detail, including the primary criteria used to make the partitioning decision and several possible partitioning algorithms. Methods for the estimation of the partitioning criteria were examined, and sources of difficulty in estimation were identified. In as many cases as was possible, possible problems and difficulties inherent in the problem were identified, and solutions suggested. While this chapter has constructed a framework for a reconfigurable compiler, additional research will be needed to overcome the many of the problems identified here before a workable system can be created.

IV. Implementation Results

4.1 Introduction

The previous chapter created the framework for the reconfigurable compiler, and illustrated the key areas in the process, as well as the difficulties involved. It is often not possible to accurately gauge the intricacies of a problem without attempting to implement a solution. For this reason, a partial implementation of a reconfigurable compiler was constructed to illustrate how the reconfigurable compiler would operate and to identify additional problems.

The implementation described in this chapter focuses on the estimation and extraction tasks. In many respects, estimation is the key to the whole partitioning process, as accurate estimates are essential for speedup to be achieved. Unfortunately, accurate estimation of hardware performance and cost is difficult, given the nature of the architecture of the FPGA. While the outer loop of the partitioning process can be used to update the estimates to more accurately reflect the intermediate implementations, routing is such a time consuming process that the synthesis step should be performed as few times as possible. Every effort should be made to estimate performance and cost as accurately as possible on the first attempt.

Extraction is implemented to demonstrate how blocks identified by the partitioner can be removed from the software code and replaced by the appropriate operating system calls. The extracted hardware code can be used to develop a behavioral hardware description language representation, which can then be synthesized automatically. This

extraction process was implemented, creating software code which can be compiled normally, and a C-like representation of the hardware functions which can be translated to behavioral VHDL or synthesized directly.

The partitioner itself was not implemented. It was shown in Chapter 3 that the partitioning problem is very complex. Runtime reconfiguration adds a level of complexity over the partitioning problem in hardware/software codesign. To efficiently utilize the reconfigurable hardware, function scheduling and bitfile grouping should be performed by the partitioner as it investigates the partition space. Although a less complex cost function was introduced as an alternative, the former approach will yield the best results. Unfortunately, this task is difficult to implement, and was beyond the scope of this thesis. To enable a more detailed examination of estimation, the implementation of the partitioner is left for future research.

4.2 The IMPACT C Compiler

The IMPACT C compiler was chosen as the basis for this implementation. IMPACT is an aggressive optimizing C compiler, under development at the University of Illinois. The compiler uses advanced techniques for code generation and optimization, such as dataflow analysis and instruction scheduling. These tools are easily accessible for use by the estimation functions. The results of each stage of the compiler are saved as intermediate files, providing a convenient means of examining the results of estimation and extraction.

IMPACT, as with many compilers, uses several intermediate formats to represent the code, as it is translated progressively from a high level representation to low level object code. In IMPACT, three primary formats are used: Pcode, Hcode, and Lcode, in successively lower levels. Pcode was chosen as the basis for this implementation. It was initially thought that Pcode was the best level at which to perform the partitioning, since the high level C constructs are still easily recognizable, and most of the compiler tools used by the estimation functions are available at this level. One of the lessons learned in this thesis was that Pcode possessed several disadvantages, and was not the best level at which to perform estimation and partitioning.

4.3 Implementation Overview

The goal for the implementation of the estimation task was to demonstrate how estimates could be made for the performance and costs of hardware implementations of C code. The estimator examines the code for hardware feasible blocks, and computes estimates of the performance and cost characteristics discussed earlier. The estimation techniques are as independent of the actual hardware as possible. The hardware estimates rapidly become dependent on the architecture of the coprocessor, its interface to the host CPU, and to the FPGAs used. Accuracy of the estimation techniques was examined, and suggestions for increased accuracy are discussed here.

The implementation of the estimator does not dynamically perform block selection, but instead works only upon loops. The code is examined and loops are identified. Hardware feasibility analysis is performed to identify and number loops which

can be implemented in hardware, and estimation is performed for eligible candidates. The estimates for each block are output in a data file, which would then be read by the partitioner and used to identify which loops should be extracted to hardware functions.

Partitioning would be performed next, followed by the extraction of the hardware functions. The extractor reads a data file containing the block numbers of the loops to be implemented as hardware functions. After removal, the resulting software code is compiled normally. The portions of the reconfigurable compiler implemented in this chapter are shown in gray in Figure 38.

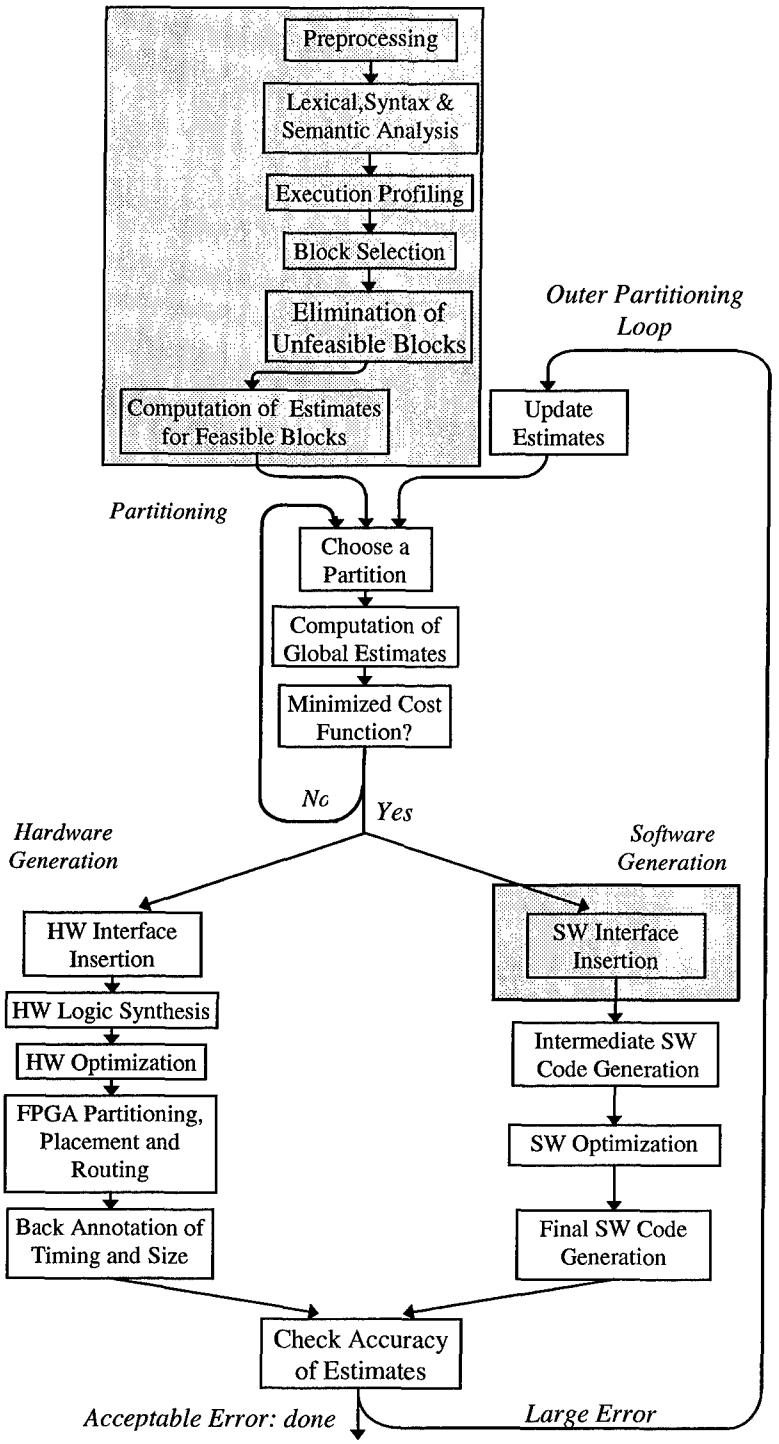

Figure 38: Implemented Tasks of the Reconfigurable Compiler

4.4 Estimation

4.4.1 Software Runtime

Software runtime is estimated for each function using a lookup based approach, as described in Chapter 3. The assembly language instructions required to implement each statement are identified, and the latencies of the instructions are summed to result in a total runtime for that statement. The total runtime for the statement is computed by multiplying this estimate by the number of times the statement was executed in the profiling runs. The sum of the estimates for all of the statements in all of the functions in the application make up the total runtime.

The implementation bases its estimates on the Sun SPARC assembly language. Each processor is unique, and a certain amount of adaptation is necessary to create accurate estimates for a given architecture. The implementation uses a parameter file to contain information about the latencies of each of the different operations (e.g. integer add, subtract, shift, floating point add, etc.) and additional information about how memory is accessed and variables are stored. The SPARC assembly language requires an addition SET instruction to set the address to load a global variable, for example, which must be taken into account when estimating the runtime of operations using the variable.

In addition to the instructions required to implement the operations in normal expression statements, additional overhead is incurred by statements such as IF-THEN, SWITCH, LOOP, CONTINUE, BREAK and GOTO. For example, *goto* statement are implemented with a branch instruction, which is followed by a NOOP due to the one

cycle branch delay of the SPARC architecture. The overhead of some statements is more complicated. Overhead for switch statements depends on the number of cases, while for loops, the overhead for initialization, condition, and iterative expressions must be included.

Software runtime estimation can be quite accurate. The bit reversal program introduced in the previous chapter was profiled and estimated. The results of the estimation are shown in Table 9, along with the actual results obtained from examination of the final assembly language output. Optimized and unoptimized code are shown.

	Estimate (cycles)	Actual (cycles)	% Error (cycles)
Bit Reversal Loop (Unoptimized)	775	677	+14.5%
Bit Reversal Loop (Optimized)	323	258	+25%
Entire Program (Unoptimized)	1400	1307	+7.1%
Entire Program (Optimized)	925	879	+5.2%
Corrected Bit Reversal Estimate (Unopt)	679		+0.3%

Table 9: Software Runtime Estimates for Bit Reversal Program

The estimates for the entire application are fairly accurate, but the estimate for the unoptimized loop is 25% larger than the actual result. Analyzing the instructions the estimator predicted would be emitted revealed that one extra MOV instruction was emitted for each of the three statements in the loop body. The estimator predicted two instructions to load the source variable for each statement: a MOV instruction to load the offset from the frame pointer, and the load instruction itself. For *Building* and *Starting*, the offsets from the frame pointer were small enough to allow the load instruction to be used with the offset contained in the constant field of the instruction, removing the need

for the extra MOV. Since the loop body is executed 32 times, the three instructions result in a 96 instruction overestimate of the actual runtime. The adjusted estimate is shown on the last line of the table.

While the bit reversal program is a trivial example, there are several lessons to be learned. First, it is possible to estimate the software runtime of unoptimized code fairly accurately at the Pcode level, before register allocation is performed and memory locations are assigned. However, since it is difficult to accurately predict whether a variable will be in a register or memory at the start of a given statement, the estimator must make assumptions. If the assumptions are incorrect, an incorrect number of instructions will be predicted. Memory addresses can change the number and types of instructions required as well.

The second lesson is that is can be very difficult to predict the effects of compiler optimization before it is performed. Optimizations such as instruction reordering and dead code elimination can greatly reduce runtime, but are performed at later stages in the compiler. With these lessons in mind, software runtime estimation should be done late in compilation, after registers have been assigned and optimization has been performed. As discussed in Chapter 3, modern superscalar compilers schedule instructions to reduce runtime. The schedules made by these compilers are usually very accurate. Software runtime estimation based upon these schedules should be much more accurate than estimates based upon Pcode.

4.4.2 Hardware Feasibility

At the same time software runtime is being performed, the statements and expressions in a function are analyzed to determine hardware feasibility. Each statement is examined for the presence of operations or data types which would invalidate the statement. Statements which contain other statements (such as compound statements and loops) are analyzed recursively, marking the child statements before a decision is made for the parent. For a parent statement to be marked as hardware feasible, all of its children must be feasible, in addition to any operations related to the statement itself, such as the conditional and iterative expressions of a loop.

Most statement and operation types are feasible for hardware implementation. GOTO and RETURN statements are marked as unfeasible, since they alter execution flow and can require complicated controls structures in hardware implementations. Operations which are marked unfeasible can include floating point operations, function calls, multiply and divide, and pointer accesses. Each type of operation has a compiler parameters which controls whether the operation is considered feasible. Floating point operations and variables, and integer multiply and divides are usually excluded because of large hardware requirements. Pointer accesses should be disallowed if the architecture does not allow the coprocessor to access memory. Finally, function calls are marked as unfeasible, since it is difficult to determine whether source code is available for the called function. If it can be determined that the called function is hardware feasible, the calling statement can be marked as feasible.

By necessity, hardware feasibility analysis is very conservative. Many blocks are eliminated which may be implemented in hardware functions. Examining several test programs revealed that many loops were eliminated from consideration due to function calls, either to library function or to other functions for which source was available. While library functions usually perform I/O or other operations which are best implemented in software, user functions may be feasible. Unfortunately, it is not possible to know whether a given function is hardware feasible until it is itself analyzed. Since functions are analyzed one at a time, a statement would have to marked as conditionally feasible, and re-evaluated once the called function is found and analyzed. Forward references make performance and cost estimation difficult as well. For this thesis, any statement containing a function call is marked as unfeasible.

Hardware feasibility is very dependent on the capabilities of the architecture. If gate capacity is large enough to implement floating point math, many digital signal processing algorithms which would have been eliminated from consideration become possible candidates. An additional group of applications would be possible if pointer accesses were allowed, even if memory addresses were limited to those explicitly sent to the coprocessor. For example, many loops access arrays with pointers instead of the element operator (e.g. *(p+loop) = 3 in place of p[loop] = 3). If the entire array is sent to the coprocessor and '*p*' only accesses the elements in that array, than the pointer operations are hardware feasible even if the coprocessor cannot access main memory. Unfortunately, it is difficult to prove that 'p' will never access an address outside of the range of the array. In many cases, it is difficult to determine that 'p' points to the array in

the first place. Unless the coprocessor can access memory independently, only those statements which operate on identifiable memory locations can be marked as feasible.

Hardware feasibility analysis can be very accurate. Since the analysis is conservative, only those code blocks which are easily realizable by the logic synthesizer are considered as candidates for partitioning. The implemented feasibility analyzer eliminated the majority of all of the loops examined, due to the factors mentioned earlier. While a smaller number of candidates reduces the work for the partitioner, it may needlessly eliminate blocks which could be productively implemented in hardware. Hardware feasibility analysis should be as permissive as possible, given the limits of the architecture.

4.4.3 Hardware Cost

Hardware cost is estimated in the manner described in Chapter 3. The implemented estimator only estimates logic block requirements. Each operation can be implemented with hardware macros, whose logic block requirements are known. Summing the requirements for all of the operations in an expression statement yields a total hardware cost for that statement. Other statements have additional hardware requirements, such as the control hardware added to For Loops to handle Break statements. This overhead is added to the total.

The calculations for the hardware cost for the bit reversal loop are shown in Table 10. The cost of each operation in the statements of the loop body are looked up and

added to the total. Overhead for the initialization conditional and iterative expressions for the loop are added next. A divide down counter is added to slow the system clock to a speed at which the loop can execute correctly. Finally, registers are created to store all of the variables used by the loop. In this case, three 32 bit registers are needed to contain *Loop, Starting,* and *Building*. The estimate is very accurate, provided that the hardware synthesizer maps code structures to hardware structures in a similar manner.

	Implementation	Logic Block Cost
Building <<= 1;	32 bit Variable Shifter	16 LBs
Building = (Starting & 0x1) | Building;	32 bit AND array, 32 bit OR array	32 LBs
Starting >>= 1;	32 bit Variable Shifter	16 LBs
Loop = 0; (Initialization Expr)	Hardwired Connections	0
Loop < 32; (Conditional Expr)	32 bit Subtractor, 32 bit comparator	34 LBs
Loop++ ; (Iterative Expr)	32 bit Adder	18 LBs
Variable Storage	3 32 bit integers	48 LBs
Divide Down Counter	Synchronous Counter	8 LBs
Total for Loop		172 LBs

Table 10: Hardware Cost Calculation for Bit Reversal Loop

The cost to implement the bit reversal as a loop is much larger than it would be to implement with a simple reversal of the wires. Wiring the most significant bit of the input to the least significant bit of the output requires no logic blocks to implement, and results in a much more efficient implementation. This illustrates the inefficiency of high level language in describing what can be very simple hardware operations. Recognizing that the loop can be replaced with a simple reversal of wire is very difficult. Automating

this process so that the reconfigurable compiler can identify the problem to be solved instead of simply following the instructions in the code is not yet possible.

Another area for improvement lies in optimization. For example, two shift operations are performed, for which the estimator uses standard hardware shifters. Since the operations shift by a constant amount however, the operations can be performed with a simply shifting of the wires by one place in the appropriate direction. The AND operation can be eliminated, since one of the operands is a constant 0x1, which effectively passes the least significant input bit to the output while forcing the others to 0. While not implemented in this thesis, this type of optimization would result in significant savings in hardware.

A more difficult optimization to implement concerns the bit widths of operations and storage. Data types in HLLs have standard bit widths, and a variable of a certain type often does not need the full number of bits to store the range of values that will be seen during execution. For example, the *loop* variable in the bit reversal loop only contains values in the range 0-31. These numbers can be represented with five bits, allowing both the register that stores the variable and all of the operations that use it to be only five bits wide as well. This reduces the amount of hardware required for the loop, and usually the runtime as well.

Implementation of this sort of optimization is nontrivial. It is often impossible to determine the range of values a variable will contain. For the bit reversal loop, *loop* is

never modified inside the loop body, is incremented, and has a static upper bound. If the upper bound was not known or the value could be changed in a way that is difficult to predict either in the iterative expression or the loop body, the range of the variable could easily exceed any tightened bounds and cause incorrect program execution. Automated reduction in the bit widths of the variables would thus have to be conservative.

The result of these inefficiencies in code implementation is that the number of code blocks which will result in speedup are reduced significantly, since hardware runtime will be larger than it would be if implemented by hand, and inflated hardware costs force fewer functions to be placed in each bitfile. The configuration time requirements are shared between fewer functions, inflating the configuration time cost for each hardware function. For the methods of synthesis described here, the logic block component of cost can be estimated fairly well. Logic reduction would result in more accurate estimates for a reasonable increase in computation. The most significant optimization, replacement of inefficient software algorithms, is much more expensive.

4.4.4 Hardware Runtime

Hardware runtime proved to be the most difficult estimate to make. For the purposes of this implementation only the combinational delay through the logic blocks was considered. Routing delay was ignored, although it can be roughly estimated with the scaling factor introduced in the previous chapter. Delay calculations are based upon the number of logic block delays for each type of operation, which are contained in the parameter file. From these values, the runtime of an expression can be determined by

adding together the logic block delays needed to perform all of the operations. Statement runtimes were computed by summing the runtimes of expressions performed in serial, and finding the maximum runtime of those performed in parallel.

The greatest source of speedup for hardware functions is parallelism. Parallelism in HLL code can be found at several levels, as discussed in Section 3.4.2.3. In practice it is difficult to take full advantage of all available parallelism. Statement level parallelism is easily achieved by performing some or all of the operations in an expression in parallel. This type of parallelism was incorporated into the estimator by analyzing the height of the Pcode tree, and was successfully implemented.

Inter-statement parallelism, which allows entire statements to be executed in parallel, proved difficult to implement at the Pcode level. Two statements can be executed at least partially in parallel if the later statement does not begin execution before all of its operands are available. Data dependency analysis can be used to determine the variables defined and used by each statement. This information, combined with the individual statement runtimes can be used to create starting and ending times for each statement. Statements are scheduled one at a time. If a statement uses a variable that is defined by a previous statement, its start time becomes the ending time for that statement. The runtime of a sequence of statements is simply the largest ending time of the group. This form of scheduling was successfully incorporated into the estimator.

Further parallelism could be achieved if the individual operations in each statement are scheduled individually. While a statement may be dependent on the results of the previous statement, operations in the later statement which do not depend on those results can be done in parallel with the first statement. Unfortunately, to perform this type of scheduling, dependency analysis must provide *define* and *use* information at the operation level of granularity, so that dependencies between individual operations can be determined. At the Pcode level of the IMPACT compiler, dependency analysis is at the statement level of granularity, making it difficult to take advantage of this type of parallelism.

Loop level parallelism was not recognized by the implemented estimator. Provided that there are no dependencies between iterations of the loop, multiple iterations can be executed in parallel. If there are dependencies, it may still be possible to pipeline the iterations of the loop to perform some of the loop iterations in parallel. This adds another level of complexity to the partitioning problem, however, since the partitioner must now decide whether to implement the loop as a single body which iterates N times, or as N bodies which iterate only once, or as some number of loop bodies in between. While a loop executed completely in parallel may be the fastest implementation of that particular loop, better use may be made of the hardware if a single body of the loop is synthesized, allowing the remaining hardware to be used for other hardware functions.

The implemented runtime estimator can estimate the delay path through the logic blocks fairly accurately. The computation of runtimes for the bit reversal loop are shown

in Table 11. As shown earlier, routing delays are difficult to estimate accurately, and more advanced estimation techniques will need to be used to determine routing effects on the overall runtime.

	Delay	Iterations	Total Runtime
Loop Body	50 ns	32	
Conditional Expr	45 ns	33	
Iterative Expr	40 ns	32	
Min Loop Clock Period	50 ns (max of above)	32	1600
Initialization Expr	0	1	0
Total Runtime			1600+0 ns

Table 11: Hardware Runtime Computation for Bit Reversal Loop

4.4.5 Hardware Communication Time

While hardware runtime is the most difficult estimate to make accurately, estimates of the hardware communication requirements are among the most accurate, second only to software runtime. As in hardware runtime, the dataflow analysis capabilities already present in the compiler are used in the estimation process. Adapting this analysis slightly allows the identification of the variables which must be sent to and retrieved from the hardware.

Calculation of the communication time requires four sets, as discussed in Chapter 3. Live In and Live Out sets were already created by the compiler's dataflow analysis. The cumulative Define and Use sets were found by unioning the define and use sets for each statement contained in the loop.

Intersecting *Cum_Use(B)* with *In(B)* and *Cum_Def(B)* with *Out(B)* yields the final sets *Comm_To(B)* and *Comm_From(B)*. The rate of communication between the CPU and the coprocessor is specified as a parameter in bytes per second. Dividing the total number of bytes to be sent by this parameter yields a maximum communication time for the block. To allow the partitioner to adjust the communication time estimates to reflect the reduction in communication that can occur when multiple hardware blocks share data, the names and individual communication times for each member of *Comm_To(B)* and *Comm_From(B)* should be output as well as the totals.

Overall, communication requirements are some of the most reliable estimates made, although there are a few sources of inefficiency. For example, the dataflow analysis implemented in the IMPACT compiler identified only which variables are used. In the case of arrays, the array itself can be identified, but not the elements. Even if only a single element of an array is needed, the entire array must be passed. To avoid sending unnecessary data between hardware and software, the dataflow analysis used in a reconfigurable compiler should track individual elements of arrays whenever possible.

4.4.6 Hardware Configuration Time

Hardware configuration time is very dependent on the nature of the FPGAs used in the coprocessor. A simple estimator was implemented by multiplying a constant parameter ("time per logic block") by the number of logic blocks estimated by the hardware cost estimator. The time per logic block parameter is derived by dividing the average number of bits per logic block by the configuration speed in bits per second. The

bit total reflects both logic block and routing configuration bits. Operating system overhead is ignored for this simple implementation. A typical value is 350 bits per logic block, and a configuration speed of 10Mbits/sec [52], yielding a parameter value of 35μs/block.

The bit reversal loop requires 124 logic blocks, a significant portion of which are used as storage for *Building, Starting,* and *Loop.* The total configuration time for this block is estimated as 4.34ms. Compared to the hardware and software runtimes of the block (2.77μs and 7.75μs) and the communication times (0.36μs and 0.12μs), the configuration time is extremely large. If the FPGA must be entirely configured for just this function, the configuration time is much higher. A hardware function as small as the bit reversal would have be called many times before the break even point is reached and speedup occurs. If only runtimes and the communication and configuration times are considered, the bit reversal function must be called 964 times before the combined hardware execution times result in a smaller total than the pure software implementation. Larger functions accomplishing more work for a larger speedup would have to be called fewer times to result in speedup.

This method of estimating the configuration is very rough, and ignored operating system overhead and the effects of partially configurable FPGAs. The reconfigurable compiler should make this estimate as accurately as possible, based upon the specifics of the hardware used in the coprocessor. For FPGAs which must be completely configured, the exact configuration times depend on the other blocks grouped into the bitfiles, and

how many times the function is to be called before reconfiguration must occur. To accurately reflect this, configuration time should really be estimated by the partitioner, during the grouping of functions into bitfiles. The estimates passed to the partitioner are rough estimates at best.

4.5 Extraction and Synthesis

After partitioning, the loops identified for hardware implementation must be removed from the software code and an interface inserted. Estimation can be performed during the same compiler pass in which profiling data is merged into the Pcode for later use in optimization. Extraction would normally be performed in the following pass. The sequence of passes in the implementation is shown in Figure 39. Partitioning is not technically a compiler pass, since it operates only the estimates and not on the source files. After the partitioning results are used to extract the hardware functions from the code, compilation of the software code can proceed normally.

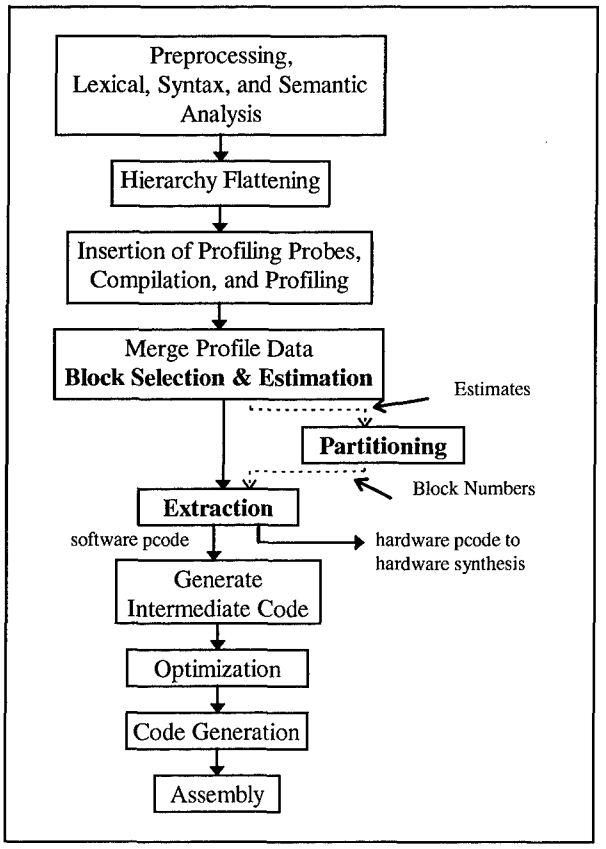

Figure 39: Sequence of Events in the Implementation

During estimation, a unique identification number was attached to each hardware feasible loop, and output as a pragma in the Pcode file. The partitioner would create as an output a file containing the numbers of the blocks to extract. During the next pass, the Pcode files are read into memory. The extractor parses the Pcode tree searching for loops which have identification numbers which match those in the partitioning file. When one is found, the loop statement is removed from the tree and replaced with an expression statement containing a function call to the operating system function controlling hardware functions. The Comm_To and Comm_From sets are used to form the parameter list for the function.

The loop statements removed from the Pcode tree are inserted into newly created functions, which are output as Pcode. An operational reconfigurable compiler would synthesize logic directly from the statement, or output a behavioral HDL function. Once all of the extracted loops are removed from the software function, it can be output normally for later compilation.

Since the partitioner was not implemented in this thesis, the estimator writes the block numbers of all of the feasible loops to the partitioning output file, so that the estimator will extract all hardware feasible loops. In the implementation, extraction was performed directly after estimation, allowing the dataflow analysis information to be used in the creation of parameter lists. If extraction is performed during a separate pass, the Comm_From and Comm_To sets will have to be recreated, or the list of parameters for each extracted hardware function passed by the partitioner.

To illustrate the proper creation of the interface, the software and hardware functions are all output as C source code, which can be compiled and tested to verify proper operation. The C code for the software portion of the bit reversal program is shown in Figure 40. Since this code was generated from an internal representation of the program halfway through compilation, some of the variables have been renamed. The loop has been removed and replaced with a call to the function HWF_main__0(). As described earlier, this function would normally be the operating system function which controls the hardware, but for demonstration purposes this is the actual hardware function itself. The input parameters, *Starting* and *Loop*, are passed by value, while the output parameter is passed by reference. *Loop* does not have to be passed, since it is initialized in the init expression of the loop, but was identified as part of the Comm_To set because of a limitation with Impact's dataflow analysis.

```
char P_fpgatestc_7_19_Input[10];
extern int main ()
{
  unsigned int P_Starting___1;
  int P_Loop___1;
  int P_Pass___1;
  unsigned int P_Building___1;
LL_1 :
        P_Starting___1 = (unsigned) 305419896;
        P_Building___1 = (unsigned) 0;
        P_Pass___1 = 34;
        printf("Please input an unsigned hexadecimal number\n");
        gets(P_fpgatestc_7_19_Input);
        sscanf(P_fpgatestc_7_19_Input,"%X",( & P_Starting___1));
        printf("The starting number is %X\n",P_Starting___1);
        HWF_main__0(P_Starting___1,P_Loop___1,( & P_Building___1));
        printf("Reversed number is %X\n",P_Building___1);
        printf("Pass through number is %d\n",P_Pass___1);
        return(0);
}
```

Figure 40: Generated C Code for the Partitioned Bit Reversal Program, Showing the Main Software Function and the Interface to the Hardware Function

The C code for the extracted hardware function is shown in Figure 41. Although the loop structure has been flattened, it is still possible to recognize the bit reversal loop. The two statements in bold are needed because a pointer to *Building* is passed instead of the value itself. To allow the code to use the same variable name used in the original function (P_Building__1), the parameter name is changed to fpga_P_Building__1, and code is inserted to dereference the pointer and copy the value into a local variable, which has the original name. At the end of the function, code is inserted to copy the local variable back into the location pointed at by the parameter. This code would be part of the operating system function, allowing the rest of the code to be synthesized as hardware.

```
extern void HWF_main__0 (P_Starting___1, P_Loop___1, fpga_P_Building___1)
  unsigned int P_Starting___1;
  int P_Loop___1;
  unsigned int *fpga_P_Building___1;
{
  unsigned int P_Building___1;
LL_1 :
        P_Building___1 = ( * fpga_P_Building___1);
        P_Loop___1 = 0;
        if (P_Loop___1 < 32) goto LL_2; else goto LL_3;
LL_2 :
        P_Building___1 <<= 1;
        P_Building___1 = ((P_Starting___1 & (unsigned) 1) | P_Building___1);
        P_Starting___1 >>= 1;
        P_Loop___1++ ;
        if (P_Loop___1 < 32) goto LL_2;
LL_3 :
        ( * fpga_P_Building___1) = P_Building___1;
        return;
}
```

Figure 41: Generated C Code for the Bit Reversal Program, Showing the Extracted Hardware Function

Although the later steps of hardware generation are time consuming, extraction and synthesis are fairly straightforward and well understood tasks. Commercial tools for logic synthesis exist, as well as tools to partition designs between multiple FPGAs.

Provided that similar tools are available for the coprocessor, hardware synthesis and routing can be automated quite effectively.

4.6 An Additional Example

The bit reversal program is a very small example used to illustrate the major concepts in the reconfigurable compiler. This section examines an actual application, used by Wright Laboratory to investigate the utility of the CHAMP I system [11]. The application is an infrared missile warning system (IRMW), typical of the sort used in modern aircraft to detect incoming missiles. The application is specified as C source, filtering large arrays containing the sensor data. Engineers at Wright Laboratory implemented a portion of the application on the CHAMP board, and compared the performance of the partitioned application to the original. The same code was used as the input to the reconfigurable compiler, and the estimates compared to the actual results.

The application consists of three types of filters: morphological, median, and ratio filters. The filters operate on a continuous stream of frames, each formed of 128x128 data samples. The median and ratio filters operate on floating point data, while the morphological filter operates on 8 bit fixed point data values. The engineers at Wright Laboratory chose to implement two parts of the morphological filter in hardware: the *dilation* filter, and the *erosion* filter.

Dilation and erosion are fairly straightforward filters. The C code for the dilation filter is shown in Figure 42. For dilation, the value of each pixel in the output array is set

to the maximum of the pixel at the same location in the input array and the eight surrounding input pixels. For erosion, the value of each pixel in the output array is set to the minimum of the pixel at the same location in the input array and the eight surrounding input pixels. Since the computation of each output pixel is independent of the other pixels, each pixel can be computed in parallel.

```
void dilation(const signed char in[ FRAME*LINE], signed char out[ FRAME*LINE])
{
  int i, j, num;
  signed char max, cur, next;

  max=in[0];
  if (in[LINE] > max)
    max=in[LINE];
  if (in[2*LINE] > max)
    max=in[2*LINE];
  cur=in[1];
  if (in[LINE+1] > cur)
    cur=in[LINE+1];
  if (in[2*LINE + 1] > cur)
    cur=in[2*LINE + 1];
  if (cur > max)
    max=cur;
  for (num=LINE+1; num < (FRAME-1)*LINE - 1; num++)
    {
      next=in[num-LINE+1];
      if (in[num+1] > next)
          next=in[num+1];
      if (in[num+LINE+1] > next)
          next=in[num+LINE+1];
      out[num]=(next > max) ? next : max;
      max=(next > cur) ? next : cur;
      cur=next;
    }
}
```

Figure 42: Source Code for the Dilation Filter in the IRMW Application

. The morphological filter is composed of two calls to each filter: dilation, erosion, erosion, and dilation, to create a final filter, which is subtracted from the input data to obtain the actual output image. The source code for the entire morphological filter is shown in Figure 43.

```
void morph(const signed char in[ FRAME*LINE], signed char out[ FRAME*LINE])
{
 static signed char tmp[FRAME*LINE];
 int i, j;

 dilation(in, out);
 erosion(out, tmp);
 erosion(tmp, out);
 dilation(out, tmp);
 for (i=0; i<FRAME; i++)
  for (j=0; j<LINE; j++)
   out[(i*LINE)+j]=in[(i*LINE)+j]-tmp[(i*LINE)+j];
}
```

Figure 43: Source Code for the Morphological Filter

One engineer manually implemented the dilation and erosion filters on the CHAMP board. The task required almost three months, using low level FPGA development tools to manually place the design onto CLBs. Each of the four filters was implemented on a separate XC4013 FPGA, leaving the final subtraction loops to be executed in software. The application was compiled normally for the attached MIPS workstation after interface code was created manually.

The hardware runtime of the dilation and erosion filters was roughly 100 times smaller than the software runtime. Communication and configuration times were not considered, since CHAMP is a static logic system and is not intended for close coupling with a workstation or runtime reconfiguration. The entire design required roughly 1400 CLBs (60% utilization of each of the four XC4013 devices). And external SRAM was used for storage of the arrays.

The same application was used as the input for the reconfigurable compiler. Four loops were identified as hardware candidates: the two loops in morph(), the loop in

dilation(), and the loop in erosion(). The time estimates for each loop were computed, and are summarized in Table 12.

Candidate	Function	SW Runtime (s)	HW Runtime (s)	Communication Time (s)	Configuration Time (s)
Loop 1	morph() inner loop	2.164E-02	1.088E-02	1.888E-01	5.040E-03
Loop 2	morph() outer loop	2.169E-02	1.088E-02	1.475E-03	7.140E-03
Loop 3	dilation()	1.032E-01	8.721E-02	9.830E-04	1.694E-02
Loop 4	erosion()	1.032E-01	8.721E-02	9.830E-04	1.722E-02

Table 12: Execution, Communication, and Configuration Time Estimates for Feasible Loops in the IRMW Application

All four loops have hardware runtimes which are smaller than the software version, although not as small as the results obtained in the manual implementation. The total runtime for each candidate loop was computed by summing the hardware runtime, configuration, and communication times. Speedup values for each block were computed, both for a single data frame, and for a continuous sequence (which effectively eliminates t_{config} by amortizing the configuration time over an infinite number of calls). The results are shown in Table 13.

Candidate	Function	SW Runtime (s)	Total T_{hw} (s)	Speedup (1 iteration)	Speedup (Asymptotic)
Loop 1	morph() inner loop	2.164E-02	0.2047	0.1057	0.1084
Loop 2	morph() outer loop	2.169E-02	1.949E-02	1.113	1.756
Loop 3	dilation()	0.1032	0.1051	0.9817	1.170
Loop 4	erosion()	0.1032	0.1054	0.9791	1.170

Table 13: Speedup Calculation for the Candidate Loops in the IRMW Application

While the speedups obtained in the dilation and erosion functions are much less than those obtained in the manual implementation, the results are positive. The manual implementation of the filters were heavily pipelined, and performed many operations in parallel. Since the implemented estimators of the reconfigurable compiler do not take advantage of loop level parallelism, each iteration of the loop body is performed sequentially, greatly increasing the required runtime.

In addition, the 100 time speedup obtained in the CHAMP tests does not account for communication of the data frames or configuration of the FPGAs. This was appropriate for the planned use of the CHAMP board, but does not fairly reflect the conditions that would be seen in a more tightly coupled system. Also, the software runtime estimates were based upon a 100 MHz SuperSPARC-II CPU, not the older MIPS processor used in the CHAMP tests. Software runtime estimates are therefore lower than the CHAMP version, resulting in lower estimated speedups. Since the estimators were based upon a non-specific hardware model, adaptation of the estimation techniques and parameters to the CHAMP model will almost certainly produce results much closer to the actual values.

While there is certainly room for improvement, the IRMW results show that the techniques are valid. The implementation of the reconfigurable compiler identified as hardware candidates the same portions of the IRMW application the human engineers found, and showed that speedup was possible. A full reconfigurable compiler would make more accurate estimates, and produce a faster implementation than was predicted

here. The partitioned design could be synthesized, placed, and routed onto the hardware in a matter of hours, compared to the 3 month manual design process. As in the early days of conventional compiler design, the performance of these automatically generated applications is likely to be worse than that of the manually created version, but there is every reason to believe that the gap will narrow as automation technology improves.

4.7 Lessons Learned

This section summarizes several of the important lessons learned in the course of implementing simple versions of the estimators and the extractor.

4.7.1 Choice of Compilation Stage

One of the first decisions that must be made in the design of a reconfigurable compiler is where to do the block selection, estimation, partitioning, and extraction. These tasks can be performed on the source programs at any stage in compilation after semantic analysis is performed. In IMPACT, the tasks can be performed on Pcode, Hcode, or Lcode. Each successive transformation of the code brings the application closer to assembly code and removes the high level structuring of the input language. For the implementation, Pcode was chosen as the source stage, primarily because it is still possible to identify the high level structure of the input. While not all of the compiler tools are available at this level, it was believed that the tools that were available would be sufficient to handle the estimation tasks.

As it turned out, block selection, estimation, partitioning, and extraction should be performed during the later stages of compilation. Software runtime estimation is much more difficult at the higher levels, before registers are assigned and optimizations are performed. Hardware runtime and cost could be more accurately estimated as well, since code optimization reduces the amount of work the hardware implementation must accomplish. Very accurate estimates of runtimes at the Pcode level would have to replicate the optimizations of the later stages. It would be more efficient to delay estimation until these effects are actually known.

While the code would be flattened and difficult to recognize at the lower levels, variable block sizes diminishes the importance of being able to identify the high level structure of the code. Software structures are examined and replaced with corresponding hardware structures. At the Pcode level, it is not difficult to identify a loop and replace it with a finite state machine. Since it is more difficult to recognize these structures at the lower levels, it will be slightly more difficult to perform hardware synthesis for the code. The hardware runtime and cost estimators would have to be modified accordingly. Ultimately, the choice of level would depend on the capabilities of the estimators used and the desired level of accuracy.

4.7.2 Dataflow Analysis

Dataflow analysis proved to be very important in the estimation of hardware runtime and communication requirements. At the Pcode level, the IMPACT compiler's dataflow analysis operates at the statement level of granularity. While this was all that

was necessary for the IMPACT compiler at this stage, a reconfigurable compiler should have dataflow information to the operation level of granularity, as shown in Section 4.4.4. This level of granularity is provided at later stages in the compilation process, and is another argument in favor of moving estimation to the later stages of the compiler. In addition, for accurate estimation of the communication requirements, dataflow analysis should provide information about the individual elements in arrays whenever possible. This would greatly reduce the overestimation of communication time which results when an entire array is sent to hardware instead of the few elements being operated upon.

V. Conclusions and Recommendations

5.1 Research Goals and Contributions

This thesis has developed a model for an automated development system for reconfigurable computers. The goal of this work was to investigate the problem of automating the application development process for these architectures, using high level language code as the input specification. To date, most of the work involved in application development for these architectures is done manually and is very expensive and time consuming. The goal of this thesis was to investigate how the process could be automated. The major contributions of this research include:

- The key tasks of a reconfigurable compiler were identified and investigated. These tasks are *block selection, estimation, partitioning*, and *synthesis*. Block selection allows code blocks of variable size to be partitioned to reduce needless communication and improve performance. Estimation of hardware performance and cost is used by the partitioner to decide which blocks of program code to implement in hardware. Partitioning determines which code blocks should be implemented as software code, and which as hardware structures. Synthesis involves the removal of these blocks from the software, insertion of a communication interface, and creation of the logic design for the FPGAs.

- The partitioning problem was examined in detail, demonstrating how the problem is similar in some ways to the partitioning problem in hardware/software codesign. The criteria used to judge individual partitions were examined:

hardware cost, software runtime, hardware runtime, communication time, and an additional factor unique to reconfigurable computing, reconfiguration time. Runtime reconfiguration adds a new dimension to the partitioning problem, effectively providing infinite hardware space at the cost of large reconfiguration times. While the reconfiguration time problem is reduced to some extent by partially configurable FPGAs, it was shown that time-based scheduling of functions onto the coprocessor can become a significant factor in partitioning.

- Methods for the estimation of the partitioning criteria were introduced. Software runtime can be estimated fairly accurately, while the hardware characteristics are more difficult. Tools available in conventional compilers such as dataflow and data dependency analysis were shown to be adaptable for use in the computation of the hardware estimates. It was shown that hardware cost is very difficult to accurately estimate, since it is based upon three different types of resources, logic blocks, I/O blocks (pins), and routing. While estimates of the logic and I/O block requirements can be made, routing estimates are difficult to make before the circuit is actually routed. Back annotation can be used to update the estimates of hardware characteristics based upon actual results of previous partitions.

- The limitations of high level language code were explained. The two most significant limitations are memory access and data width. Pointer-based operations require the coprocessor to be able to access memory independently of the CPU. This suggests that future architecture designers allow independent memory access. Data types of fixed widths were shown to cause inefficient

hardware implementations, which may be improved through optimization. In many cases, however, this is difficult to perform automatically.

- The implementation discussed in Chapter 4 provided a proof of concept for the major tasks of the reconfigurable compiler, and of estimation in particular. Each estimation task is best performed at a specific stage in compilation. It was shown that software runtime estimation is best performed later in compilation, while hardware structures may be easier to identify and estimate at earlier stages. The importance of dataflow analysis and other compiler tools was shown, and limitations identified. In particular, fine granularity in dataflow analysis was shown to be necessary for accurate hardware runtime and communication estimation.

5.2 Recommendations for Future Research

Reconfigurable computing is a very new field of research, and there are many areas which merit further study. Five specific areas were identified whose future study would improve the efficiency of the reconfigurable compiler:

- **Block Selection.** While block sizes of fixed granularity may be used in partitioning, more efficient results may be achieved if blocks were formed to take full advantage of natural boundaries in the code. The largest effect is on communication times, which would be significantly reduced since data does not have to be sent back and forth between the CPU and the coprocessor. Further

research into methods for block selection should discover the best methods for dynamically grouping code into blocks.

- **Estimation techniques.** In many ways, the effectiveness of the whole reconfigurable compiler relies on accurate estimation of the performance and cost characteristics used in the partitioning. Since FPGA routing is very time consuming, estimates of each partition's performance characteristics must be made without the benefit of post-routing back annotation of timing and cost. If the estimates are not accurate, the resulting partition will not perform optimally, and may even be slower than a pure software partitioning of the code.

- **Partitioning.** While the partitioning problem in reconfigurable computing is similar in some ways to that of hardware/software codesign, runtime reconfiguration adds a new dimension to the problem. Whereas codesign deals with a "fixed bin" in which hardware functions may be implemented, a reconfigurable system has effectively infinite hardware, limited only by reconfiguration overhead. Scheduling of hardware functions becomes important, as does the grouping of functions into bitfiles to reduce the configuration overhead as much as possible. New algorithms should be developed to take these factors into account as the partition space is searched.

- **The limitations of HLLs for hardware description.** Chapters 3 and 4 showed how high level languages have difficulty describing hardware. Hardware generated from high level language descriptions is much less efficient in terms of both performance and size than hardware generated from an HDL. As in the early

days of compiler design, applications developed by hand will have better performance than applications developed with automated tools. Further research may make it possible to reduce some of the inefficiencies of HLL code with optimization and other techniques.

- **Operating System model**. The operating system model described in this thesis was merely one of many possibilities. The operating system must handle communication between the CPU and the hardware and configuration of the FPGAs, including tracking of which hardware functions are present on the hardware at any given instant. If possible, the operating system should even be allowed to perform preloading of configurations. There is currently no detailed model of how these tasks should be performed.

5.3 Conclusion

The implementation of a reconfigurable compiler is a complicated task, involving the combination of ideas from many areas of research. The problem is similar in some ways to the automation of the codesign of embedded systems, but runtime reconfiguration and other factors add levels of complexity to the problem which must be addressed. Many of the components of a reconfigurable compiler can be adapted from these other areas.

The characteristics of the FPGAs and HLLs which would provide the greatest benefits when used for reconfigurable computing also create difficulties in application development. Partitioning depends on accurate pre-routing estimation of runtime and

cost characteristics of each partition, which is difficult to accomplish given the architecture of the FPGA. This problem may be reduced as FPGA architectures evolve to support reconfigurable computing, and routing algorithms improve to make their results more predictable. High level language support for reconfigurable computers opens up a huge range of applications which may benefit from implementation on these systems, and removes the need for an engineer to manually implement applications with an HDL. Although HLL derived hardware is currently less efficient than HDL derived hardware, it may be sufficient to provide significant speedups for many applications, particularly those which possess large amounts of parallelism.

While further research must be done before an truly efficient reconfigurable compiler can be created, the groundwork has been laid. This thesis has created a framework upon which such a reconfigurable compiler would be based. Although, a reconfigurable compiler could be created using current knowledge, the accuracy of its estimates and performance of its compiled applications may be insufficient. Further research will improve the accuracy of the partitioning, and should provide an effective means for normal software to take advantage of reconfigurable hardware.

Bibliography

1. Abbott, A. Lynn, Athanas, Peter, Tarmaster, Adit, "Accelerating Image Filters Using a Custom Computing Machine", FPGAs for Fast Board Development and Reconfigurable Computing. Proceedings of the SPIE 2607, Billingham, WA, 1995.

2. Aho, Alfred V., Sethi, Ravi, Ullman, Jeffrey D., Compilers: Principles, Techniques, and Tools, Addison-Wesley Publishing, Reading, MA. 1986.

3. Albaharna, Osama T., Cheung, Peter Y.K., Clarke, Thomas J., "On the Viability of FPGA-Integrated Coprocessors," IEEE Symposium on FPGAs for Custom Computing Machines, Napa CA, April 1996.

4. Athanas, Peter M., Abott A. Lynn, "Real-Time Image Processing on a Custom Computing Platform." IEEE Computer, Vol. 28, Num. 2. February 1995.

5. Athanas, Peter M., Silverman, Harvey F., "Processor Reconfiguration Through Instruction Set Metamorphosis," IEEE Computer, Vol. 26, Num. 3, March 1993.

6. Atmel Corporation, "AT6000 Series Configuration".

7. Barros, E., Rosensteil, W., "A Method for Hardware/Software Partitioning," Proceedings of Compeuro, IEEE Computer Society Press, Los Alamitos, CA, 1992.

8. Benner, T., Ernst, R., Könenkamp, I., Schüler, P., Schaub, H.-C., "A Prototyping System for Verification and Evaluation in Hardware-Software Cosynthesis," Proceedings of the 6th International Workshop on Rapid System Prototyping. Chapel Hill, NC, 1995.

9. Benner,T., Henkel, J., Ernst, R., "Internal Representation of Embedded Hardware/Software Systems," Codes/CASHE Workshop. Innsbrück, FRG, 1993.

10. Bertin, P., Roncin, D., Vuillemin, J., "Introduction to Programmable Active Memories," DEC Paris Research Laboratory #3, 1989.

11. Box, Brian, "Configurable Hardware Algorithm Mappable Processor (CHAMP)", Lockheed Sanders, Nashua NH, 1995.

12. Box, Brian, Nieznanski, John, "Common Processor Element Packaging for CHAMP," Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines, April 1995.

13. Brown, Stephen, "FPGA Architectural Research: A Survey", IEEE Design & Test of Computers, Vol. 13, Num. 4, Winter 1996.

14. Buell, D.A., Arnold, J.M., Kleinfelder, W.J., eds., Splash 2: FPGAs for Custom Computing Machines. IEEE Computer Society Press, Los Alamitos, CA, 1995.

15. Carlson, Steve, Introduction to HDL-Based Design Using VHDL, Synopsys, inc., Mountaint View, CA, 1990.

16. Casselman, Steve, Thornburg, Michael, Schewel, John, "Hardware Object Programming on the EVC1: a Reconfigurable Computer", FPGAs for Fast Board Development and Reconfigurable Computing. Proceedings of the SPIE 2607, Billingham, WA, 1995.

17. DeHon, Andre, "DPGA-Coupled Microprocessors: Commodity ICs for the early 21st Century," Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines, April 1994.

18. DeMicheli, Giovanni, "Computer-Aided Hardware-Software Codesign", IEEE Micro, Vol. 14, Num. 4, August 1994.

19. Ernst, Rolf, Henkel, Jörg, Benner, Thomas, "Hardware-Software Cosynthesis for Microcontrollers," IEEE Design & Test of Computers, Vol. 10, Num. 4, December 1993.

20. Fawcett, Bradly, "Field Programmable Gate Arrays and Reconfigurable Computing," FPGAs for Fast Board Development and Reconfigurable Computing. Proceedings of the SPIE 2607. Billingham, WA, 1995.

21. Galloway, David, "The Transmogrifier C Hardware Description Language and Compiler for FPGAs", Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines. April 1995.

22. Gokhale, M., et al., "SPLASH: A Reconfigurable Linear Logic Array," Proceedings of the International Conference on Parallel Processing, August 1990.

23. Gupta, Rajesh K., Coelho, Claudionor N., De Micheli, Giovanni, "Program Implementation Schemes for Hardware-Software Systems," IEEE Computer, Vol. 27, Num. 1, January 1994.

24. Gupta, Rajesh K., De Micheli, Giovanni, "Hardware-Software Cosynthesis for Digital Systems," IEEE Design & Test of Computers, Vol. 10, Num. 3, September 1993.

25. Harbison, Samuel P., Steele, Guy L., C: A Reference Manual, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1987.

26. Henkel, J., Benner, T., Ernst, R., "Hardware Generation and Partitioning Effects in the COSYMA System," Proceedings of the 2nd IEEE/ACM International Workshop on Hardware/Software Codesign. Cambridge, MA, 1993.

27. Henkel, Jörg, Ernst, Rolf, "A Path-Based Technique for Estimating Hardware Runtime in HW/SW-Cosynthsis," IEEE Proceedings of the 8th International Symposium on System Level Synthesis. Cannes, France, 1995.

28. Herrmann, D., Henkel, J., Ernst, R., "An Approach to the Adaptation of Estimated Cost Parameters in the COSYMA System," Proceedings of the 3rd IEEE International Workshop on Hardware/Software Codesign. 1994.

29. Hesener, Alfred, "Cache Logic: One FPGA per Board," Electronic Engineering, Vol. 67, Num. 3, March 1995.

30. Institute of Electrical and Electronic Engineers, IEEE-STD 1076-1993 IEEE Standard VHDL Language Reference Manual, IEEE Press, New York, 1993. {30}

31. Iseli, Christian, Sanchez, Eduardo, "A C++ Compiler for FPGA Custom Execution Units Synthesis", Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines. April 1995.

32. Ismail, Tarek Ben, Jerraya, Ahmed Amine. "Synthesis Steps and Design Models for Codesign," IEEE Computer. Vol. 28, Num.2. February 1995.

33. Jantsch, Axel, Ellervee, Peeter, Öberg, Johnny, Hermani, Ahmed, Tenhunen, Hannu, "Hardware/Software Partitioning and Minimizing Memory Interface Traffic," Proceedings of the European Design Automation Conference. Grenoble, 1994.

34. Kalavade, Asawaree, Lee, Edward A., "A Hardware-Software Codesign Methodology for DSP Applications," IEEE Design & Test of Computers, Vol. 10, Num. 3, September 1993.

35. Kumar, Sanjaya, Aylor, James H., Johnson, Barry W., Wulf, Wm. A., "A Framework for Hardware/Software Codesign," IEEE Computer, Vol. 26, Num. 12, December 1993.

36. Lawman, Gary, Configuring FPGAs Over a Processor Bus, Xilinx Application Note, Xilinx, inc., San Jose, CA., 1994.

37. Liu, Philip S., Mowle, Frederic J., "Techniques of Program Execution with a Writable Control Store," IEEE Transactions on Computers, Vol. C-27 Num. 9, September 1978.

38. Marsh, William D. II, Evaluation of VHDL as a Hardware/Software Codesign Tool. MS Thesis, AFIT/GCS/ENG/94D-17. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1994.

39. Olukotun, Kunle, Helaihel, Rachid, Levitt, Jeremy, Ramirez, Ricardo, "A Software-Hardware Cosynthesis Approach to Digital System Simulation", IEEE Micro, Vol. 14, Num. 4, August 1994.

40. Rado, Ted, "Counting Gates in Programmable Logic," Electronic Engineering, Vol. 67, Num. 12, December 1995.

41. Razdan, R., Smith, M.D., "A High-Performance Microarchitecture with Hardware-Programmable Functional Units," Micro 27, November 1994.

42. Rozenblitz, Jerzy, Buchenreider, Klaus, "Codesign: An Overview," Codesign: Computer-Aided Software/Hardware Engineering, Rozenblitz, Jerzy, Buchenreider, Klaus, editors, IEEE Press, Piscataway, NJ 1995.

43. Srivastava, M.B., Brodersen, R.W., "Rapid-Prototyping of Hardware and Software in a Unified Framework," Proceedings of the International Conference on Computer-Aided Hardware Design, IEEE Computer Society Press, Los Alamitos, CA, 1991.

44. Thomas, Donald E., Adams, Jay K., Schmit, Herman. "A Model and Methodology for Hardware-Software Codesign," IEEE Design & Test of Computers, Vol. 10, Num. 3, September 1993.

45. Vahid, Frank, Gajski, Daniel D., "Incremental Hardware Estimation During Hardware/Software Functional Partitioning," IEEE Transactions on VLSI Systems, Vol. 3, Num. 3, September 1995.

46. Vahid, Frank, Gong, Jie, Gajski, Daniel, D., "A Binary-Constraint Search Algorithm for Minimizing Hardware During Hardware/Software Partitioning," Proceedings of the European Design Automation Conference. Grenoble, 1994.

47. Van den Bout, D., et al., "Anyboard: An FPGA-Based Reconfigurable System," IEEE Design and Test of Computers, Vol. 9, Num. 3, September 1992.

48. Wirthlin, M.J., Hutchings, B.L., Gilson, K.L., "The NanoProcessor: A Low Resource Reconfigurable Processor," Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines, Napa, CA, April 1995.

49. Wirthlin, Michael J., Hutchings, Brad L., "A Dynamic Instruction Set Computer," Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines, April 1995.

50. Wittig, Ralph D., Chow, Paul, "OneChip: An FPGA Processor with Reconfigurable Logic," IEEE Symposium on FPGAs for Custom Computing Machines, April 1996.

51. Woo, Nam S., Dunlop, Alfred E., Wolf, Wayne, "Codesign from Cospecification," IEEE Computer, Vol. 27, Num 1, January 1994.

52. Xilinx, inc., "The Programmable Logic Data Book," San Jose, CA., 1994.

53. Xilinx, inc., "XC4000E Field Programmable Gate Array Family Preliminary Product Specifications," San Jose, CA, Sept. 1995.

54. Ye, W., Ernst, R., Benner, T., Henkel, J., "Fast Timing Analysis for Hardware-Software Co-Synthesis," 1993 IEEE International Conference on Computer Design, IEEE Computer Society Press, Los Alamitos, CA, 1993.

55. York, Trevor, "Survey of Field Programmable Logic Devices", Microprocessors and Microsystems, Vol. 17, Num 4, September 1993.

Vita

George R. Roelke IV[redacted]. He graduated from Carlisle Senior High School in Carlisle, Pennsylvania, in 1991, and received his undergraduate education at the Georgia Institute of Technology in Atlanta, Georgia. Graduating from Georgia Tech in 1995 with a Bachelor of Computer Engineering degree, he was assigned to the Air Force Institute of Technology. Here, he pursued the Master of Science in Computer Engineering degree, and began his research in Reconfigurable Architectures. His next assignment is to the National Air Intelligence Center, also at Wright-Patterson AFB.

[redacted]

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	March 1997	Master's Thesis

4. TITLE AND SUBTITLE

A FRAMEWORK FOR AN AUTOMATED COMPILATION SYSTEM FOR RECONFIGURABLE ARCHITECTURES

5. FUNDING NUMBERS

6. AUTHOR(S)

George Roelke, 2Lt, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology
2750 P Street, WPAFB OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT/GE/ENG/97M-01

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Keith Anthony
NAIC/TACC
WPAFB OH 45433

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The advent of the Field Programmable Gate Array has allowed the implementation of runtime reconfigurable computer systems. These systems are capable of configuring their hardware to provide custom hardware support for software applications. Since these architectures can be reconfigured during operation, they are able to provide hardware support for a variety of applications, without removal from the system. The Air Force is currently investigating reconfigurable architectures for avionics and signal processing applications.

This thesis investigates the problem of automating the application development process for reconfigurable architectures. The lack of automated development support is a major limiting factor in the use of these systems. This thesis creates a framework for a reconfigurable compiler, which automatically implements a single high level language specification as a reconfigurable hardware/software application. The major tasks in the process are examined, and possible methods for implementation are investigated. A prototype reconfigurable compiler has been developed to demonstrate the feasibility of important concepts, and to uncover additional areas of difficulty.

14. SUBJECT TERMS

Reconfigurable Computing, Custom Computing Machines, Automated Development, Compilers, Partitioning, Runtime Reconfiguration

15. NUMBER OF PAGES

209

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

GENERAL INSTRUCTIONS FOR COMPLETING SF 298

The Report Documentation Page (RDP) is used in announcing and cataloging reports. It is important that this information be consistent with the rest of the report, particularly the cover and title page. Instructions for filling in each block of the form follow. It is important to ***stay within the lines*** to meet ***optical scanning requirements***.

Block 1. Agency Use Only *(Leave blank)*.

Block 2. Report Date. Full publication date including day, month, and year, if available (e.g. 1 Jan 88). Must cite at least the year.

Block 3. Type of Report and Dates Covered. State whether report is interim, final, etc. If applicable, enter inclusive report dates (e.g. 10 Jun 87 - 30 Jun 88).

Block 4. Title and Subtitle. A title is taken from the part of the report that provides the most meaningful and complete information. When a report is prepared in more than one volume, repeat the primary title, add volume number, and include subtitle for the specific volume. On classified documents enter the title classification in parentheses.

Block 5. Funding Numbers. To include contract and grant numbers; may include program element number(s), project number(s), task number(s), and work unit number(s). Use the following labels:

C	- Contract	**PR**	- Project
G	- Grant	**TA**	- Task
PE	- Program Element	**WU**	- Work Unit Accession No.

Block 6. Author(s). Name(s) of person(s) responsible for writing the report, performing the research, or credited with the content of the report. If editor or compiler, this should follow the name(s).

Block 7. Performing Organization Name(s) and Address(es). Self-explanatory.

Block 8. Performing Organization Report Number. Enter the unique alphanumeric report number(s) assigned by the organization performing the report.

Block 9. Sponsoring/Monitoring Agency Name(s) and Address(es). Self-explanatory.

Block 10. Sponsoring/Monitoring Agency Report Number. *(If known)*

Block 11. Supplementary Notes. Enter information not included elsewhere such as: Prepared in cooperation with...; Trans. of...; To be published in.... When a report is revised, include a statement whether the new report supersedes or supplements the older report.

Block 12a. Distribution/Availability Statement. Denotes public availability or limitations. Cite any availability to the public. Enter additional limitations or special markings in all capitals (e.g. NOFORN, REL, ITAR).

- **DOD** - See DoDD 5230.24, "Distribution Statements on Technical Documents."
- **DOE** - See authorities.
- **NASA** - See Handbook NHB 2200.2.
- **NTIS** - Leave blank.

Block 12b. Distribution Code.

- **DOD** - Leave blank.
- **DOE** - Enter DOE distribution categories from the Standard Distribution for Unclassified Scientific and Technical Reports.
- **NASA** - Leave blank.
- **NTIS** - Leave blank.

Block 13. Abstract. Include a brief *(Maximum 200 words)* factual summary of the most significant information contained in the report.

Block 14. Subject Terms. Keywords or phrases identifying major subjects in the report.

Block 15. Number of Pages. Enter the total number of pages.

Block 16. Price Code. Enter appropriate price code *(NTIS only)*.

Blocks 17. - 19. Security Classifications. Self-explanatory. Enter U.S. Security Classification in accordance with U.S. Security Regulations (i.e., UNCLASSIFIED). If form contains classified information, stamp classification on the top and bottom of the page.

Block 20. Limitation of Abstract. This block must be completed to assign a limitation to the abstract. Enter either UL (unlimited) or SAR (same as report). An entry in this block is necessary if the abstract is to be limited. If blank, the abstract is assumed to be unlimited.